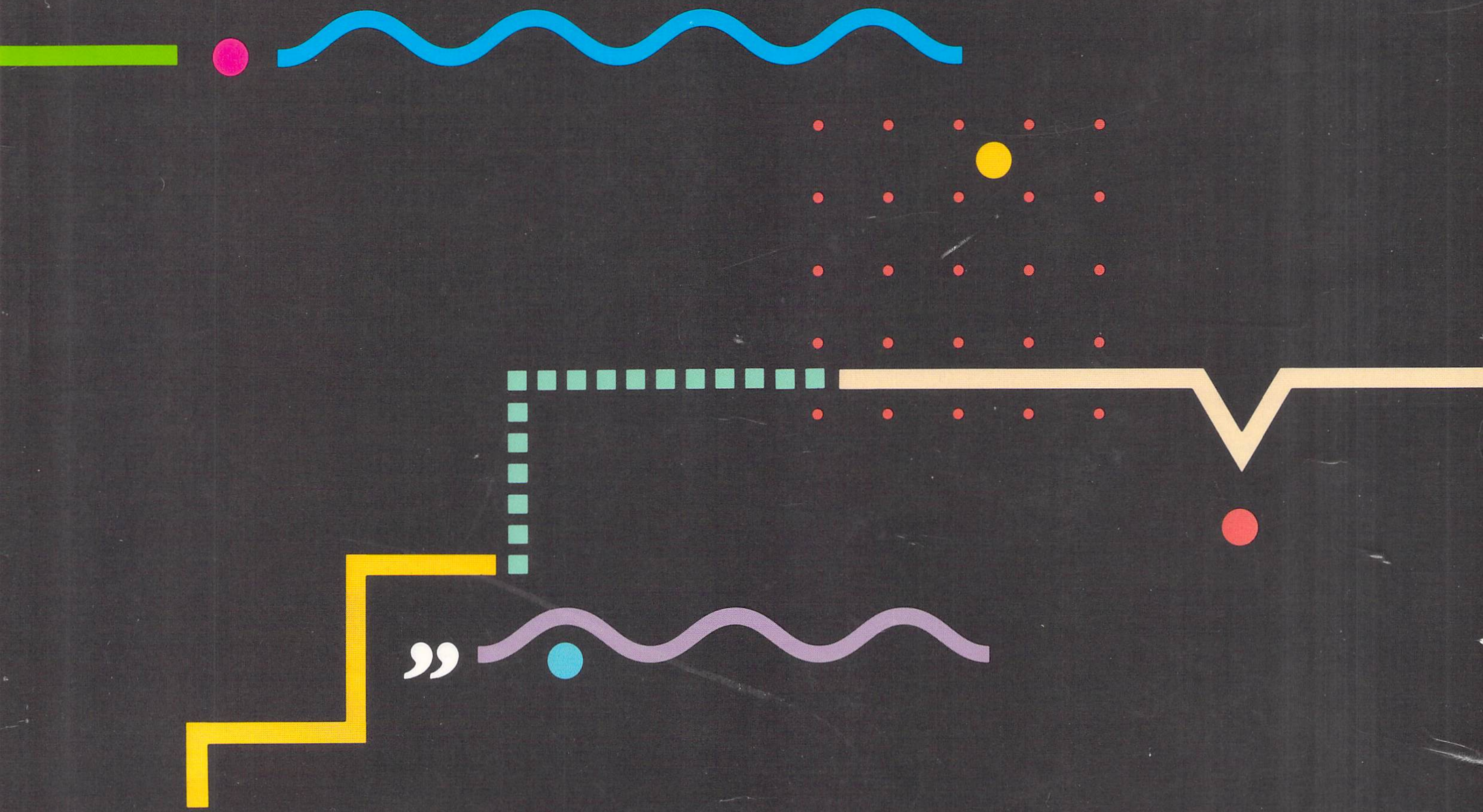
AMIGA® ROM Kernel Reference Manual

DEVICES



AMIGA TECHNICAL REFERENCE SERIES

COMMODORE-AMIGA, INC.

THIRD EDITION

AMIGA®
ROM Kernel Reference Manual
Devices

Third Edition

Commodore-Amiga, Inc.

AMIGA TECHNICAL REFERENCE SERIES

Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts   Menlo Park, California   New York
Don Mills, Ontario   Wokingham, England   Amsterdam   Bonn
Sydney   Singapore   Tokyo   Madrid   San Juan
Paris   Seoul   Milan   Mexico City   Taipei

Contributors:
Dan Baker, Bruce Barrett, Mark Barton, Steve Beats, Dave Berezowski, Ray Brand, Bob Burns, Peter Cherna, Eric Cotton, Susan Deyl, Sam Dicker, Ken Farinsky, Stuart Ferguson, Andy Finkel, Darren Greenwald, Paul Higginbottom, Larry Hildenbrand, Randell Jesup, David Junod, Neil Katin, Joe Katz, Kevin Klop, Adam Levin, Dave Lucas, Dale Luck, Jim Mackraz, R.J. Mical, Bryce Nesbitt, John Orr, Bob Pariseau, Rob Peck, Tom Pohorsky, Carl Sassenrath, Stan Shepard, Michael Sinz, Nancy Rains, Chris Raymond, Tom Rokicki, Jez San, Carolyn Scheppner, Leo Schwab, Darius Taghavy, Ewout Walraven, Bart Whitebook and Rob Wyesham.

Third edition by:
Mark Ricci

Cover designer:
Hannus Design Associates

Commodore item number: 363099-02

123456789-AL-9594939291
First printing, September 1991
ISBN 0-201-56775-X

CONTENTS

Preface

The Amiga® Technical Reference Series is the official guide to programming Commodore's Amiga computers. This revised edition of the *Amiga ROM Kernel Reference Manual: Devices* provides detailed information about the Amiga's I/O subsystems. It has been updated for Release 2 (Kickstart V36 and up) of the Amiga operating system, however, most of the material and example programs are also compatible with version 1.3.

This book is intended for the following audiences:

- Novice Amiga programmers who want to try out features of the Amiga devices without writing full-blown applications.
- Experienced programmers new to the Amiga.
- Amiga programmers and developers who want to use the devices in an application.

It is assumed that the reader can program in C or at least understand it.

Here is a brief overview of the contents:

Chapter 1, *Introduction to Amiga System Devices.* An introduction to the concept of an Amiga system device, the device interface, and how to perform I/O using the devices.

Chapter 2, *Audio Device.* The Amiga audio device allows you to play music and make sounds. Two example programs are included.

Chapter 3, *Clipboard Device.* The clipboard device is a central facility for sharing information between applications. The chapter covers the types of clipboard data and the proper ways to use the clipboard. Two example programs are included plus an extensively commented module of support functions for the programs.

Chapter 4, *Console Device.* The console device is the text-oriented interface for Amiga windows. The chapter lists the escape sequences used for console windows and the types of console windows. An example program is included.

Chapter 5, *Gameport Device.* The gameport device manages the various pointing devices you plug into the mouse/joystick connectors. The chapter discusses the types of pointing devices, the protocol for using the device and includes an example program.

Chapter 6, *Input Device.* The input device collects input event information and passes this on to the operating system. The chapter covers this interaction between the various input sources of the system, tells how to create your own input events and includes two example programs.

Chapter 7, *Keyboard Device.* The keyboard device is the Amiga keyboard manager. The chapter covers how to read the keyboard at a low level and also how to program system reset (Ctrl-Amiga-Amiga) handlers. Three example programs are included.

Chapter 8, *Narrator Device.* The narrator device is the voice of the Amiga. This chapter explains how to use the narrator device and the translator library, how to write phonetic strings for the device, and discusses the technical aspects of computer generated speech in thorough, but understandable terms. Two example programs are included.

Chapter 9, *Parallel Device.* The parallel device manages the Amiga parallel port. Two example programs are included.

Chapter 10, *Printer Device.* The printer device translates character streams into printer specific sequences. The chapter covers how to use the printer device and how to write your own printer driver. It contains two example programs and two complete printer drivers.

Chapter 11, *SCSI Device.* The SCSI device provides the Small Computer System Interface for the Amiga. The chapter covers how to send Amiga specific and SCSI specific commands to SCSI devices. An example program is included.

Chapter 12, *Serial Device.* The serial device manages the Amiga serial port. Three example programs are included.

Chapter 13, *Timer Device.* The timer device an interface to the Amiga's internal clocks. The chapter explains the types of clocks and clock units. Four example programs are included.

Chapter 14, *Trackdisk Device.* The trackdisk device controls the Amiga disk drives. The chapter covers how to use the drives at a high-level (formatted reads and writes) and low-level (raw reads and writes). An example program is included.

Chapter 15, *Resources.* The Amiga resources are a collection of low-level interfaces to special Amiga hardware. The chapter covers the general resource interface and how to use all seven resources. Example code is included for all but one of the resources.

Appendix A, *IFF, Interchange File Format.* IFF is the standardized file format of the Amiga. This appendix introduces IFF, covers five of the IFF types, lists the official FORM and Chunk names that are reserved and in use and how to register new ones. IFF include files, link modules, example programs and utilities are included.

Appendix B, *Example Device.* This appendix contains the assembly code for an Amiga device for all those who want to create their own custom software I/O device.

Appendix C, *Amiga Floppy Boot Process and Physical Layout.* This appendix lists the method used to read the boot block of a floppy and how the data is arranged in the boot block.

The other manuals in this series are the *Amiga User Interface Style Guide*, an application design specification and reference work for Amiga programmers, the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*, an alphabetically organized reference of ROM function summaries and Amiga system include files, the *Amiga ROM Kernel Reference Manual: Libraries*, a work consisting of tutorial-style chapters on the use of each Amiga system library, and the *Amiga Hardware Reference Manual*, a detailed description of the Amiga's hardware components.

chapter one

INTRODUCTION TO AMIGA SYSTEM DEVICES

The Amiga system devices are software engines that provide access to the Amiga hardware. Through these devices, a programmer can operate a modem, spin a disk drive motor, time an event, speak to a user and blast a trumpet sound in beautiful, living stereo. Yet, for all that variety, the programmer uses each device in the same basic manner.

Amiga System Devices	
Audio	Controls the use of the audio hardware
Clipboard	Manages the cutting and pasting of common data blocks
Console	Provides the text-oriented user interface.
Gameport	Controls the two mouse/joystick ports.
Input	Processes input from the gameport and keyboard devices.
Keyboard	Controls the keyboard.
Narrator	Produces the Amiga synthesized speech.
Parallel	Controls the parallel port.
Printer	Converts a standard set of printer control codes to printer specific codes.
SCSI	Controls the Small Computer Standard Interface hardware.
Serial	Controls the serial port.
Timer	Provides timing functions to measure time intervals and send interrupts.
Trackdisk	Controls the Amiga floppy disk drives.

What is a Device?

An Amiga device is a software module that accepts commands and data and performs I/O operations based on the commands it receives. In most cases, a device interacts with either internal or external hardware. Generally, an Amiga device runs as a separate task which is capable of processing your commands while your application attends to other things.

Device I/O is based on the EXEC messaging system. The philosophy behind the devices is that I/O operations should be consistent and uniform. You print a file in the same manner as you play an audio sample, i.e., you send the device in question a WRITE command and the address of the buffer holding the data you wish to write.

The result is that the interface presented to the programmer is essentially device independent and accessible from any computer language. This greatly expands the power the Amiga computer brings to the programmer and, ultimately, to the user.

Devices support two types of commands: Exec standard commands like READ and WRITE, and device specific commands like the trackdisk device MOTOR command which controls the floppy drive motor. The Exec standard commands are supported by most Amiga devices. You should keep in mind, however, that supporting standard commands does not mean that all devices execute them in *exactly* the same manner.

This manual contains a chapter about each of the Amiga devices. The chapters cover how you use a device and the commands it supports. In addition, the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* contains expanded explanations of the commands and the include files for each device, and the *Amiga ROM Kernel Reference Manual: Libraries* contains chapters on Exec. The command explanations list the data, flags, and other information required by a device to execute a command. The Exec chapters provide detailed discussions of its operation. Both are very useful manuals to have on your desk when you are programming the devices.

Accessing a Device

Accessing a device requires obtaining a message port, allocating memory for a specialized message packet called an I/O request, setting a pointer to the message port in the I/O request, and finally, establishing the link to the device itself by opening it. An example of how to do this will be provided later in this chapter.

The message port is used by the device to return messages to you. A message port is obtained by calling the **CreateMsgPort()** or **CreatePort()** function. You must delete the message port when you are finished by calling the **DeleteMsgPort()** or **DeletePort()** function.

For pre-V36 versions of the operating system (before Release 2.0), use the amiga.lib functions **CreatePort()** and **DeletePort()**; for V36 and higher, use the Exec functions **CreateMsgPort()** and **DeleteMsgPort()**. **CreatePort()** and **DeletePort()** are upward compatible, you can use them with V36/V37; **CreateMsgPort()** and **DeleteMsgPort()** are not backward compatible, however.

The I/O request is used to send commands and data from your application to the device. The I/O request consists of fields used to hold the command you wish to execute and any parameters it requires. You set up the fields with the appropriate information and send it to the device by using Exec I/O functions.

At least four methods exist for creating an I/O request:

- Declaring it as a structure. The memory required will be allocated at compile time.
- Declaring it as a pointer and calling the **AllocMem()** function. You will have to call the **FreeMem()** function to release the memory when you are done.
- Declaring it as a pointer and calling the **CreateExtIO()** function. This function not only allocates the memory for the request, it also puts the message port in the I/O request. You will have to call the **DeleteExtIO()** function to delete the I/O request when you are done. This is the pre-V36 method (used in 1.3 and earlier versions of the operating system), but is upward compatible.
- Declaring it as a pointer and calling the **CreateIORequest()** function. This function not only allocates the memory for the request, it also puts the message port in the I/O request. You will have to call the **DeleteIORequest()** function to delete the I/O request when you are done. This is the V36/V37 method; it is not backwards compatible.

The message port pointer in the I/O request tells the device where to respond with messages for your application. You must set a pointer to the message port in the I/O request if you declare it as a structure or allocate memory for it using **AllocMem()**.

The device is opened by calling the **OpenDevice()** function. In addition to establishing the link to the device, **OpenDevice()** also initializes fields in the I/O request. **OpenDevice()** has this format:

```
return = OpenDevice(device_name,unit_number,(struct IORequest *)IORequest,flags)
```

where:

- **device_name** is one of the following NULL-terminated strings for system devices:

audio.device	keyboard.device	serial.device
clipboard.device	narrator.device	timer.device
console.device	parallel.device	trackdisk.device
gameport.device	printer.device	
input.device	scsi.device	

- **unit_number** refers to one of the logical units of the device. Devices with one unit always use unit 0. Multiple unit devices like the trackdisk device and the timer device use the different units for specific purposes. The device chapters discuss the units in detail.
- **IORequest** is the structure discussed above. Some of the devices have their own I/O requests defined in their include files and others use standard I/O requests, (**IOStdReq**). The device chapters list the I/O request that each device requires.
- **flags** are bits set to indicate options for some of the devices. This field is set to zero for devices which don't accept options when they are opened. The device chapters and autodocs list the flags values and uses.
- **return** is an indication of whether the **OpenDevice()** was successful with zero indicating success. Never assume that a device will successfully open. Check the return value and act accordingly.

Zero Equals Success for OpenDevice(). Unlike most Amiga system functions, **OpenDevice()** returns zero for success and a device-specific error value for failure.

Using a Device

Once a device has been opened, you use it by passing the I/O request to it. When the device processes the I/O request, it acts on the information the I/O request contains and returns a reply message, i.e., the I/O request, to the reply port specified in the I/O request when it is finished. The I/O request is passed to a device using one of three functions, **DoIO()**, **SendIO()** and **BeginIO()**. They take only one argument: the I/O request you wish to pass to the device.

- **DoIO()** is a synchronous function. It will not return until the device has responded to the I/O request.
- **SendIO()** is an asynchronous function. It can return immediately, but the I/O operation it initiates may take a short or long time. Using **SendIO()** requires you to monitor the message port for a return message from the device. In addition, some devices do not actually respond asynchronously even though you called **SendIO()**; they will return when the I/O operation is finished.
- **BeginIO()** is commonly used to control the quick I/O bit when sending an I/O request to a device. When the quick I/O flag (IOF_QUICK) is set in the I/O request, a device is allowed to take certain shortcuts in performing and completing a request. If the request can complete immediately, the device will not return a reply message and the quick I/O flag will remain set. If the request cannot be completed immediately, the QUICK_IO flag will be clear. **DoIO()** normally requests quick I/O; **SendIO()** does not.

DoIO() and **SendIO()** are most commonly used.

An I/O request typically has three fields set for every command sent to a device. You set the command itself in the **io_Command** field, a pointer to the data for the command in the **io_Data** field, and the length of the data in the **io_Length** field.

```
SerialIO->IOSer.io_Length   = sizeof(ReadBuffer);
SerialIO->IOSer.io_Data  = ReadBuffer;
SerialIO->IOSer.io_Command  = CMD_READ;
SendIO((struct IORequest *)SerialIO);
```

Commands consist of two parts—a prefix and the command word separated by an underscore, all in upper case. The prefix indicates whether the command is an Exec or device specific command. All Exec commands have **CMD** as the prefix. They are defined in the include file *exec/io.h*.

Amiga Exec Commands

CMD_CLEAR	CMD_READ	CMD_STOP
CMD_FLUSH	CMD_RESET	CMD_WRITE
CMD_INVALID	CMD_START	CMD_UPDATE

You should not assume that a device supports all Exec commands. Always check the documentation before attempting to use one of them.

Device specific command prefixes vary with the device.

Amiga System Device Command Prefixes and Examples

Device	Prefix	Example
Audio	ADCMD	ADCMD_ALLOCATE
Clipboard	CBD	CBD_POST
Console	CD	CD_ASKKEYMAP
Gameport	GPD	GPD_SETCTYPE
Input	IND	IND_SETMPORT
Keyboard	KBD	KBD_READMATRIX
Narrator	no device specific commands	
Parallel	PDCMD	PDCMD_QUERY
Printer	PRD	PRD_PRTCOMMAND
SCSI	HD	HD_SCSICMD
Serial	SDCMD	SDCMD_BREAK
Timer	TR	TR_ADDREQUEST
Trackdisk	TD and ETD	TD_MOTOR/ETD_MOTOR

Each device maintains its own I/O request queue. When a device receives an I/O request, it either processes the request immediately or puts it in the queue because one is already being processed. After an I/O request is completely processed, the device checks its queue and if it finds another I/O request, begins to process that request.

Synchronous vs. Asynchronous Requests

As stated above, you can send I/O requests to a device synchronously or asynchronously. The choice of which to use is largely a function of your application.

Synchronous requests use the **DoIO()** function. **DoIO()** will not return control to your application until the I/O request has been satisfied by the device. The advantage of this is that you don't have to monitor the message port for the device reply because **DoIO()** takes care of all the message handling. The disadvantage is that your application will be tied up while the I/O request is being processed, and should the request not complete for some reason, **DoIO()** will not return and your application will hang.

Asynchronous requests use the **SendIO()** and **BeginIO()** functions. Both return to your application almost immediately after you call them. This allows you to do other operations, including sending more I/O requests to the device.

> *Do Not Touch!* When you use **SendIO()** or **BeginIO()**, the I/O request you pass to the device and any associated data buffers should be considered read-only. Once you send it to the device, you must *not* modify it in any way until you receive the reply message from the device or abort the request (though you must still wait for a reply). Any exceptions to this rule are documented in the autodoc for the device.

Sending multiple asynchronous I/O requests to a device can be tricky because devices require them to be unique and initialized. This means you can't use an I/O request that's still in the queue, but you need the fields which were initialized in it when you opened the device. The solution is to copy the initialized I/O request to another I/O request(s) before sending anything to the device.

Regardless of what you do while you are waiting for an asynchronous I/O request to return, you need to have some mechanism for knowing when the request has been done. There are two basic methods for doing this.

The first involves putting your application into a wait state until the device returns the I/O request to the message port of your application. You can use the **WaitIO()**, **Wait()** or **WaitPort()** function to wait for the return of the I/O request.

WaitIO() not only waits for the return of the I/O request, it also takes care of all the message handling functions. This is very convenient, but you can pay for this convenience: your application will hang in the unlikely event that the I/O request does not return.

Wait() waits for a signal to be sent to the message port. It will awaken your task when the signal arrives, but you are responsible for all of the message handling.

WaitPort() waits for the message port to be non-empty. It returns a pointer to the message in the port, but you are responsible for all of the message handling.

The second method to detect when the request is complete involves using the **CheckIO()** function. **CheckIO()** takes an I/O request as its argument and returns an indication of whether or not it has been completed. When **CheckIO()** returns the completed indication, you will still have to remove the I/O request from the message port.

I/O Request Completion

A device will set the **io_Error** field of the I/O request to indicate the success or failure of an operation. The indication will be either zero for success or a non-zero error code for failure. There are two types of error codes: Exec I/O and device specific. Exec I/O errors are defined in the include file *exec/errors.h*; device specific errors are defined in the include file for each device. You should always check that the operation you requested was successful.

The exact method for checking **io_Error** can depend on whether you use **DoIO()** or **SendIO()**. In both cases, **io_Error** will be set when the I/O request is returned, but in the case of **DoIO()**, the **DoIO()** function itself returns the same value as **io_Error**.

This gives you the option of checking the function return value:

```
SerialIO->IOSer.io_Length   = sizeof(ReadBuffer);
SerialIO->IOSer.io_Data  = ReadBuffer;
SerialIO->IOSer.io_Command  = CMD_READ;
if (DoIO((struct IORequest *)SerialIO);
    printf("Read failed.  Error: %ld\n",SerialIO->IOSer.io_Error);
```

Or you can check **io_Error** directly:

```
SerialIO->IOSer.io_Length   = sizeof(ReadBuffer);
SerialIO->IOSer.io_Data  = ReadBuffer;
SerialIO->IOSer.io_Command  = CMD_READ;
DoIO((struct IORequest *)SerialIO);

if (SerialIO->IOSer.io_Error)
    printf("Read failed.  Error: %ld\n",SerialIO->IOSer.io_Error);
```

Keep in mind that checking **io_Error** is the *only* way that I/O requests sent by **SendIO()** can be checked.

Testing for a failed I/O request is a minimum step, what you do beyond that depends on your application. In some instances, you may decide to resend the I/O request, and in others, you may decide to stop your application. One thing you'll almost always want to do is to inform the user that an error has occurred.

> *Exiting The Correct Way.* If you decide that you must prematurely end your application, you should deallocate, release, give back and let go of everything you took to run the application. In other words, you should exit gracefully.

Ending Device Access

You end device access by reversing the steps you took to access it. This means you close the device, deallocate the I/O request memory and delete the message port. In that order!

Closing a device is how you tell Exec that you are finished using a device and any associated resources. This can result in housecleaning being performed by the device. However, before you close a device, you might have to do some housecleaning of your own.

A device is closed by calling the **CloseDevice()** function. The **CloseDevice()** function does not return a value. It has this format:

```
CloseDevice(IORequest)
```

where **IORequest** is the I/O request used to open the device.

You should not close a device while there are outstanding I/O requests, otherwise you can cause major and minor problems. Let's begin with the minor problem: memory. If an I/O request is outstanding at the time you close a device, you won't be able to reclaim the memory you allocated for it.

The major problem: the device will try to respond to the I/O request. If the device tries to respond to an I/O request, and you've deleted the message port (which is covered below), you will probably crash the system.

One solution would be to wait until all I/O requests you sent to the device return. This is not always practical if you've sent a few requests and the user wants to exit the application immediately.

In that case, the only solution is to abort and remove any outstanding I/O requests. You do this with the functions **AbortIO()** and **WaitIO()**. They must be used together for cleaning up. **AbortIO()** will abort an I/O request, but will not prevent a reply message from being sent to the application requesting the abort. **WaitIO()** will wait for an I/O request to complete and remove it from the message port. This is why they must be used together.

> *Be Careful With AbortIO().* Do not **AbortIO()** an I/O request which has *not* been sent to a device. If you do, you may crash the system.

After the device is closed, you must deallocate the I/O request memory. The exact method you use depends on how you allocated the memory in the first place. For **AllocMem()** you call **FreeMem()**, for **CreateExtIO()** you call **DeleteExtIO()**, and for **CreateIORequest()** you call **DeleteIORequest()**. If you allocated the I/O request memory at compile time, you naturally have nothing to free.

Finally, you must delete the message port you created. You delete the message port by calling **DeleteMsgPort()** if you used **CreateMsgPort()**, or **DeletePort()** if you used **CreatePort()**.

Here is the checklist for gracefully exiting:

1. Abort any outstanding I/O requests with **AbortIO()**
2. Wait for the completion of any outstanding or aborted I/O requests with **WaitIO()**.
3. Close the device with **CloseDevice()**.
4. Release the I/O request memory with either **DeleteIORequest()**, **DeleteExtIO()** or **FreeMem()** (as appropriate).
5. Delete the message port with **DeleteMsgPort()** or **DeletePort()**.

Devices With Functions

Some devices, in addition to their commands, provide functions which can be directly called by applications. These functions are documented in the device specific FD files and Autodocs of the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* and the device chapters of this manual.

Devices with functions behave much like Amiga libraries, i.e., you set up a base address pointer and call the functions as offsets from the pointer. (See the "Exec: Libraries" chapter of the *Amiga ROM Kernel Reference Manual: Libraries*.)

The procedure for accessing a device's functions is as follows:

- Declare the device base address variable in the global data area. The name of the base address can be found in the device's FD file.
- Create a message port using one of the previously discussed methods if you haven't already done so.
- Create an I/O request using one of the previously discussed methods if you haven't already done so. Remember to set the message port pointer in the I/O request if necessary.
- Call **OpenDevice()**, passing the I/O request if you haven't already done so. When you do this, the device returns a pointer to its base address in the **io_Device** field of the I/O request structure. Consult the include file for the structure you are using to determine the full name of the **io_Device** field. The base address is only valid while the device is open.
- Set the device base address variable to the pointer returned in the **io_Device** field.

We will use the timer device to illustrate the above method. The name of the timer device base address is listed in its FD file as "TimerBase."

```
#include <devices/timer.h>

struct Library *TimerBase;    /* device base address pointer */

struct MsgPort *TimerMP;      /* message port pointer */
struct timerequest *TimerIO;  /* I/O request pointer */

    /* Create the message port */
if (TimerMP=CreatePort(NULL,NULL))
    {
        /* Create the I/O request */
    if (TimerIO = (struct timerequest *)
        {           CreateExtIO(TimerMP,sizeof(struct timerequest)))
            /* Open the timer device */
        if (!(OpenDevice(TIMERNAME,UNIT_MICROHZ,TimerIO,0)))
            {
            /* Set up pointer for timer functions */
            TimerBase = (struct Library *)TimerIO->tr_node.io_Device;

            ... use functions ...

            /* Close the timer device */
            CloseDevice(TimerIO);
            }

        /* Delete the I/O request */
        DeleteExtIO(TimerIO);
        }

    /* Delete the message port */
    DeletePort(TimerMP);
    }
```

Example Device Programs

The following short programs are examples of how to use a device. Both send the serial device command **SDCMD_QUERY** to the serial device to determine the status of the serial device lines and registers. The first program is for pre-V36 versions of the operating system (before Release 2) and the second is for V36 and higher. You may use the pre-V36 version with V36 and higher, but you may not use the V36 version with older systems.

The programs differ in the way they create the message port and I/O request. The pre-V36 version uses the amiga.lib functions **CreatePort()** to create the message port and **CreateExtIO()** to create the I/O request; the V36 version uses the Exec functions **CreateMsgPort()** to create the message port and **CreateIORequest()** to create the I/O request. Those are the only differences.

DEVICE USAGE EXAMPLE (PRE-V36)

```
/*
 * Pre_V36_Device_Use.c
 *
 * This is an example of using the serial device.
 * First, we will attempt to create a message port with CreatePort()
 * Next, we will attempt to create the I/O request with CreateExtIO()
 * Then, we will attempt to open the serial device with OpenDevice()
 * If successful, we will send the SDCMD_QUERY command to it
 * and reverse our steps.
 * If we encounter an error at any time, we will gracefully exit.
 *
 * Compile with SAS C 5.10  lc -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/serial.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>
```

```
#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void main(void)
{
struct MsgPort *SerialMP;       /* pointer to our message port */
struct IOExtSer *SerialIO;      /* pointer to our I/O request */

    /* Create the message port */
if (SerialMP=CreatePort(NULL,NULL))
    {
        /* Create the I/O request */
    if (SerialIO = (struct IOExtSer *)CreateExtIO(SerialMP,sizeof(struct IOExtSer)))
        {
            /* Open the serial device */
        if (OpenDevice(SERIALNAME,0,(struct IORequest *)SerialIO,0L))

            /* Inform user that it could not be opened */
            printf("Error: %s did not open\n",SERIALNAME);
        else
            {
            /* device opened, send query command to it */
            SerialIO->IOSer.io_Command  = SDCMD_QUERY;
            if (DoIO((struct IORequest *)SerialIO))

                /* Inform user that query failed */
                printf("Query  failed. Error - %d\n",SerialIO->IOSer.io_Error);
            else
                /* Print serial device status - see include file for meaning */
                printf("\n\tSerial device status: %x\n\n",SerialIO->io_Status);

            /* Close the serial device */
            CloseDevice((struct IORequest *)SerialIO);
            }
        /* Delete the I/O request */
        DeleteExtIO(SerialIO);
        }
    else
        /* Inform user that the I/O request could be created */
        printf("Error: Could create I/O request\n");

    /* Delete the message port */
    DeletePort(SerialMP);
    }
else
    /* Inform user that the message port could not be created */
    printf("Error: Could not create message port\n");
}
```

DEVICE USAGE EXAMPLE (KICKSTART V36 AND UP)

```
/*
 * V36_Device_Use.c
 *
 * This is an example of using the serial device.
 * First, we will attempt to create a message port with CreateMsgPort()
 * Next, we will attempt to create the I/O request with CreateIORequest()
 * Then, we will attempt to open the serial device with OpenDevice()
 * If successful, we will send the SDCMD_QUERY command to it
 * and reverse our steps.
 * If we encounter an error at any time, we will gracefully exit.
 *
 * Compile with SAS C 5.10  lc -cfistq -v -y -L
 *
 * Requires Kickstart V36 or greater.
 *
 * Run from CLI only
 */
```

```
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/serial.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void main(void)
{
struct MsgPort *SerialMP;        /* pointer to our message port */
struct IOExtSer *SerialIO;       /* pointer to our I/O request */

    /* Create the message port */
if (SerialMP=CreateMsgPort())
    {
        /* Create the I/O request */
    if (SerialIO = CreateIORequest(SerialMP,sizeof(struct IOExtSer)))
        {
            /* Open the serial device */
        if (OpenDevice(SERIALNAME,0,(struct IORequest *)SerialIO,0L))

            /* Inform user that it could not be opened */
            printf("Error: %s did not open\n",SERIALNAME);
        else
            {
            /* device opened, send query command to it */
            SerialIO->IOSer.io_Command  = SDCMD_QUERY;
            if (DoIO((struct IORequest *)SerialIO))

                /* Inform user that query failed */
                printf("Query  failed. Error - %d\n",SerialIO->IOSer.io_Error);
            else
                /* Print serial device status - see include file for meaning */
                printf("\n\tSerial device status: %x\n\n",SerialIO->io_Status);

            /* Close the serial device */
            CloseDevice((struct IORequest *)SerialIO);
            }
        /* Delete the I/O request */
        DeleteIORequest(SerialIO);
        }
    else
        /* Inform user that the I/O request could be created */
        printf("Error: Could create I/O request\n");

    /* Delete the message port */
    DeleteMsgPort(SerialMP);
    }
else
    /* Inform user that the message port could not be created */
    printf("Error: Could not create message port\n");
}
```

chapter two
AUDIO DEVICE

The Amiga has four hardware audio channels—two of the channels produce audio output from the left audio connector, and two from the right. These channels can be used in many ways. You can combine a right and a left channel for stereo sound, use a single channel, or play a different sound through each of the channels to create four-part harmony.

About Amiga Audio

Most personal computers that produce sound have hardware designed for one *specific* synthesis technique. The Amiga computer uses a very general method of digital sound synthesis that is quite similar to the method used in digital hi-fi components and state-of-the-art keyboard and drum synthesizers.

For programs that can afford the memory, playing sampled sounds gives you a simple and very CPU-efficient method of sound synthesis. A sampled sound is a table of numbers which represents a sound digitally. When the sound is played back by the Amiga, the table is fed by a DMA channel into one of the four digital-to-analog converters in the custom chips. The digital-to-analog converter converts the samples into voltages that can be played through amplifiers and loudspeakers, reproducing the sound.

On the Amiga you can create sound data in many other ways. For instance, you can use trigonometric functions in your programs to create the more traditional sounds—sine waves, square waves, or triangle waves—by using tables that describe their shapes. Then you can combine these waves for richer sound effects by adding the tables together. Once the data are entered, you can modify them with techniques described below. For information about the limitations of the audio hardware and suggestions for improving system efficiency and sound quality, refer to the *Amiga Hardware Reference Manual.*

Some commands enable your program to co-reside with other programs using the audio device at the same time. Programs can co-reside because the audio device handles allocation of audio channels and arbitrates among programs competing for the same resources. When properly used, this allows many programs to use the audio device simultaneously.

The audio device commands help isolate the programmer from the idiosyncrasies of the custom chip hardware and make it easier to use. But you can also produce sound on the Amiga by directly accessing the hardware registers if you temporarily lock out other users first. For certain types of sound synthesis, this is more CPU-efficient.

DEFINITIONS

Terms used in the following discussions may be unfamiliar. Some of the more important ones are defined below.

Amplitude
The height of a waveform, which corresponds to the amount of voltage or current in the electronic circuit.

Amplitude modulation
A means of producing special audio effects by using one channel to alter the amplitude of another.

Channel
One "unit" of the audio device.

Cycle
One repetition of a waveform.

Frequency
The number of times per second a cycle repeats.

Frequency modulation
A means of producing special audio effects by using one channel to affect the period of the waveform produced by another channel.

Period
The time elapsed between the output of successive sound samples, in units of system clock ticks.

Precedence
Priority of the user of a sound channel.

Sample
Byte of audio data, one of the fixed-interval points on the waveform.

Waveform
Graph that shows a model of how the amplitude of a sound varies over time—usually over one cycle.

Audio Device Commands and Functions

Command	Operation
ADCMD_ALLOCATE	Allocate one or more of the four audio channels.
ADCMD_FINISH	Abort the current write request on one or more of the channels. Can be done immediately or at the end of the current cycle.
ADCMD_FREE	Free one or more audio channels.
ADCMD_LOCK	Lock one or more audio channels.
ADCMD_PERVOL	Change the period and volume for writes in progress. Can be done immediately or at the end of the cycle.
ADCMD_SETPREC	Set the allocation precedence of one or more channels.
ADCMD_WAITCYCLE	Wait for the current write cycle to complete on a single channel. Returns at the end of the cycle or immediately if no cycle is active on the channel.
CMD_FLUSH	Purge all write cycles and waitcycles (in-progress and queued) for one or more channels.
CMD_READ	Return a pointer to the I/O block currently writing on a single channel.
CMD_RESET	Reset one or more channels their initialized state. All active and queued requests will be aborted.
CMD_START	Resume writes to one or more channels that were stopped.
CMD_STOP	Stop any write cycle in progress on one or more channels.
CMD_WRITE	Start a write cycle on a single channel.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the audio device. If in progress, it is stopped immediately, otherwise it is removed from the queue.
BeginIO()	Initiate a command and return immediately (asynchronous request).
CheckIO()	Determine the current state of an I/O request.
CloseDevice()	Relinquish use of the audio device.
OpenDevice()	Obtain use of the audio device.
Wait()	Wait for a signal from the audio device.
WaitPort()	Wait for the audio message port to receive a message.

Exec Support Functions as Used in This Chapter

AllocMem()	Allocate a block of memory.
CreatePort()	Create a signal message port for reply messages from the audio device. Exec will signal a task when a message arrives at the reply port.
DeletePort()	Delete the message port created by **CreatePort()**.
FreeMem()	Free a block of previously allocated memory.

Device Interface

The audio device operates like the other Amiga I/O devices. To make sound, you first open the audio device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

Audio device commands use an extended I/O request block named **IOAudio** to send commands to the audio device. This is the standard **IORequest** block with some extra fields added at the end.

```
struct IOAudio
{
    struct IORequest ioa_Request;    /* I/O request block.  See exec/io.h.      */
    WORD   ioa_AllocKey;             /* Alloc. key filled in by audio device   */
    UBYTE  *ioa_Data;                /* Pointer to a sample or allocation mask */
    ULONG  ioa_Length;               /* Length of sample or allocation mask.   */
    UWORD  ioa_Period;               /* Sample playback speed                  */
    UWORD  ioa_Volume;               /* Volume of sound                        */
    UWORD  ioa_Cycles;               /* # of times to play sample. 0=forever.  */
    struct Message ioa_WriteMsg;     /* Filled in by device - usually not used */
};
```

See the include file *devices/audio.h* for the complete structure definition.

OPENING THE AUDIO DEVICE

Before you can use the audio device, you must first open it with a call to **OpenDevice()**. Four primary steps are required to open the audio device:

- Create a message port using **CreatePort**. Reply messages from the device must be directed to a message port.
- Allocate memory for an extended I/O request structure of type **IOAudio** using **AllocMem()**.
- Fill in **io_Message.mn_ReplyPort** with the message port created by **CreatePort**.
- Open the audio device. Call **OpenDevice()**, passing **IOAudio**.

```
struct MsgPort   *AudioMP;          /* Define storage for port pointer */
struct IOAudio   *AudioIO;          /* Define storage for IORequest pointer */

if (AudioMP = CreatePort(0,0) )
    {
    AudioIO = (struct IOAudio *)
              AllocMem(sizeof(struct IOAudio), MEMF_PUBLIC | MEMF_CLEAR);
    if (AudioIO)
        {
        AudioIO->ioa_Request.io_Message.mn_ReplyPort  = AudioMP;
        AudioIO->ioa_AllocKey                         = 0;
        }

    if (OpenDevice(AUDIONAME,0L,(struct IORequest *)AudioIO,0L) )
        printf("%s did not open\n",AUDIONAME);
```

A special feature of the **OpenDevice()** function with the audio device allows you to automatically allocate channels for your program to use when the device is opened. This is convenient since you *must* allocate one or more channels before you can produce sound.

This is done by setting **ioa_AllocKey** to zero, setting **ioa_Request.io_Message.mn_Node.ln_Pri** to the appropriate precedence, setting **io_Data** to the address of a channel combination array, and setting **ioa_Request.ioa_Length** to a non-zero value (the length of the channel combination array).

The audio device will attempt to allocate channels just as if you had sent the ADCMD_ALLOCATE command (see below). If the allocation fails, the **OpenDevice()** call will return immediately.

If you want to allocate channels at some later time, set the **ioa_Request.ioa_Length** field of the **IOAudio** block to zero when you call **OpenDevice()**. For more on channel allocation and the ADCMD_ALLOCATE command, see the section on "Allocation and Arbitration" below.

```
UBYTE chans[] = {1,2,4,8};  /* get any of the four channels */

if (AudioIO)
    {
    AudioIO->ioa_Request.io_Message.mn_ReplyPort  = AudioMP;
    AudioIO->ioa_AllocKey                         = 0;
    AudioIO->ioa_Request.io_Message.mn_Node.ln_Pri= 120;
    AudioIO->ioa_Data                             = chans;
    AudioIO->ioa_Length                           = sizeof(chans);
    }

if (OpenDevice(AUDIONAME,0L,(struct IORequest *)AudioIO,0L) )
    printf("%s did not open\n",AUDIONAME);
```

AUDIO DEVICE COMMAND TYPES

Commands for audio use can be divided into two categories: allocation/arbitration commands and hardware control commands.

There are four allocation/arbitration commands. These do not actually produce any sound. Instead they manage and arbitrate the audio resources for the many tasks that may be using audio in the Amiga's multitasking environment.

ADCMD_ALLOCATE - Reserves an audio channel for your program to use.
ADCMD_FREE - Frees an audio channel.
ADCMD_SETPREC - Changes the precedence of a sound in progress.
ADCMD_LOCK - Tells if a channel has been stolen from you.

The hardware control commands are used to set up, start, and stop sounds on the audio device:

CMD_WRITE - The main command. Starts a sound playing.
ADCMD_FINISH - Aborts a sound in progress.
ADCMD_PERVOL - Changes the period (speed) and volume of a sound in progress.
CMD_FLUSH - Clears the audio channels.
CMD_RESET - Resets and initializes the audio device.
ADCMD_WAITCYCLE - Signals you when a cycle finishes.
CMD_STOP - Temporarily stops a channel from playing.
CMD_START - Restarts an audio channel that was stopped.
CMD_READ - Returns a pointer to the current **IOAudio** request.

SCOPE OF AUDIO COMMANDS

Most audio commands can operate on multiple channels. The exceptions are ADCMD_WAITCYCLE, CMD_WRITE and CMD_READ, which can only operate on one channel at a time. You specify the channel(s) that you want to use by setting the appropriate bits in the **ioa_Request.io_Unit** field of the **IOAudio** block. If you send a command for a channel that you do not own, your command will be ignored. For more details, see the section on "Allocation and Arbitration" below.

AUDIO AND SYSTEM I/O FUNCTIONS

BeginIO()

All the commands that you can give to the audio device should be sent by calling the **BeginIO()** function. This differs from other Amiga devices which generally use **SendIO()** or **DoIO()**. You should not use **SendIO()** or **DoIO()** with the audio device because these functions clear some special flags used by the audio device; this might cause audio to work incorrectly under certain circumstances. To be safe, you should always use **BeginIO()** with the audio device.

Wait() and WaitPort()

These functions can be used to put your task to sleep while a sound plays. **Wait()** takes a wake-up mask as its argument. The wake-up mask is usually the **mp_SigBit** of a **MsgPort** that you have set up to get replies back from the audio device.

WaitPort() will put your task to sleep while a sound plays. The argument to **WaitPort()** is a pointer to a **MsgPort** that you have set up to get replies back from the audio device.

Wait() and **WaitPort()** will not remove the message from the reply port. You must use **GetMsg()** to remove it.

You must always use **Wait()** or **WaitPort()** to wait for I/O to finish with the audio device.

AbortIO()

This function can be used to cancel requests for ADCMD_ALLOCATE, ADCMD_LOCK, CMD_WRITE, or ADCMD_WAITCYCLE. When used with the audio device, **AbortIO()** always succeeds.

CLOSING THE AUDIO DEVICE

An **OpenDevice()** must eventually be matched by a call to **CloseDevice()**.

All I/O requests must be complete before **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()**:

```
AbortIO((struct IORequest *)AudioIO);  /* Abort any pending requests */
WaitPort(AudioMP);                     /* Wait for abort message */
GetMsg(AudioMP);                       /* Get abort message */
CloseDevice((struct IORequest *)AudioIO);
```

CloseDevice() performs an ADCMD_FREE command on any channels selected by the **ioa_Request.io_Unit** field of the **IOAudio** request. This means that if you close the device with the same **IOAudio** block that you used to allocate your channels (or a copy of it), the channels will be automatically freed.

If you allocated channels with multiple allocation commands, you cannot use this function to close all of them at once. Instead, you will have to issue one ADCMD_FREE command for each allocation that you made. After issuing the ADCMD_FREE commands for each of the allocations, you can call **CloseDevice()**.

A Simple Audio Example

The Amiga's audio software has a complex allocation and arbitration system which is described in detail in the sections below. At this point, though, it may be helpful to look at a simple audio example:

```
/*
 * Audio.c
 *
 * Audio example
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <devices/audio.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <graphics/gfxbase.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>
#include <clib/graphics_protos.h>

#include <stdlib.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct GfxBase *GfxBase;

/*----------------------------------------------------------*/
/* The whichannel array is used when we allocate a channel.  */
/* It tells the audio device which channel we want. The code */
/* is 1 =channel0, 2 =channel1, 4 =channel2, 8 =channel3.    */
/* If you want more than one channel, add the codes up.      */
/* This array says "Give me channel 0. If it's not available */
/* then try channel 1; then try channel 2 and then channel 3 */
/*----------------------------------------------------------*/
UBYTE           whichannel[] = { 1,2,4,8 };

void main(int argc, char **argv)
{
struct IOAudio *AudioIO;      /* Pointer to the I/O block for I/O commands   */
struct MsgPort *AudioMP;      /* Pointer to a port so the device can reply */
struct Message *AudioMSG;     /* Pointer for the reply message             */
ULONG           device;
BYTE           *waveptr;               /* Pointer to the sample bytes     */
LONG            frequency = 440;       /* Frequency of the tone desired   */
LONG            duration  = 3;         /* Duration in seconds             */
LONG            clock     = 3579545;   /* Clock constant, 3546895 for PAL */
LONG            samples   = 2;         /* Number of sample bytes          */
LONG            samcyc    = 1;         /* Number of cycles in the sample  */

/*------------------------------------------------------------------------*/
/* Ask the system if we are PAL or NTSC and set clock constant accordingly */
/*------------------------------------------------------------------------*/
GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0L);
if (GfxBase == 0L)
    goto killaudio;
if (GfxBase->DisplayFlags & PAL)
    clock = 3546895;        /* PAL clock */
else
    clock = 3579545;        /* NTSC clock */

if (GfxBase)
    CloseLibrary((struct Library *) GfxBase);
```

```
/*-------------------------------------------------------------------------*/
/*  Create an audio I/O block so we can send commands to the audio device   */
/*-------------------------------------------------------------------------*/
AudioIO = (struct IOAudio *)
           AllocMem( sizeof(struct IOAudio),MEMF_PUBLIC | MEMF_CLEAR);
if (AudioIO == 0)
    goto killaudio;
printf("IO block created...\n");

/*-----------------------------------------------------------------*/
/* Create a reply port so the audio device can reply to our commands */
/*-----------------------------------------------------------------*/
AudioMP = CreatePort(0,0);
if (AudioMP == 0)
    goto killaudio;
printf("Port created...\n");

/*----------------------------------------------------------------------*/
/* Set up the audio I/O block for channel allocation:                   */
/* ioa_Request.io_Message.mn_ReplyPort is the address of a reply port.  */
/* ioa_Request.io_Message.mn_Node.ln_Pri sets the precedence (priority) */
/*   of our use of the audio device. Any tasks asking to use the audio  */
/*   device that have a higher precedence will steal the channel from us.*/
/* ioa_Request.io_Command is the command field for I/O.                 */
/* ioa_Request.io_Flags is used for the I/O flags.                      */
/* ioa_AllocKey will be filled in by the audio device if the allocation */
/*   succeeds. We must use the key it gives for all other commands sent.*/
/* ioa_Data is a pointer to the array listing the channels we want.     */
/* ioa_Length tells how long our list of channels is.                   */
/*----------------------------------------------------------------------*/
AudioIO->ioa_Request.io_Message.mn_ReplyPort   = AudioMP;
AudioIO->ioa_Request.io_Message.mn_Node.ln_Pri = 0;
AudioIO->ioa_Request.io_Command                = ADCMD_ALLOCATE;
AudioIO->ioa_Request.io_Flags                  = ADIOF_NOWAIT;
AudioIO->ioa_AllocKey                          = 0;
AudioIO->ioa_Data                              = whichannel;
AudioIO->ioa_Length                            = sizeof(whichannel);
printf("I/O block initialized for channel allocation...\n");

/*----------------------------------------------*/
/* Open the audio device and allocate a channel  */
/*----------------------------------------------*/
device = OpenDevice(AUDIONAME,0L, (struct IORequest *) AudioIO ,0L);
if (device != 0)
    goto killaudio;
printf("%s opened, channel allocated...\n",AUDIONAME);

/*---------------------------------------------*/
/* Create a very simple audio sample in memory. */
/* The sample must be CHIP RAM                  */
/*---------------------------------------------*/
waveptr = (BYTE *)AllocMem( samples , MEMF_CHIP|MEMF_PUBLIC);
if (waveptr == 0)
    goto killaudio;
waveptr[0] =  127;
waveptr[1] = -127;
printf("Wave data ready...\n");

/*---------------------------------------------------------------*/
/* Set up audio I/O block to play a sample using CMD_WRITE.       */
/* The io_Flags are set to ADIOF_PERVOL so we can set the        */
/*    period (speed) and volume with the our sample;              */
/* ioa_Data points to the sample; ioa_Length gives the length */
/* ioa_Cycles tells how many times to repeat the sample           */
/* If you want to play the sample at a given sampling rate,       */
/* set ioa_Period = clock/(given sampling rate)                   */
/*---------------------------------------------------------------*/
AudioIO->ioa_Request.io_Message.mn_ReplyPort =AudioMP;
AudioIO->ioa_Request.io_Command              =CMD_WRITE;
AudioIO->ioa_Request.io_Flags                =ADIOF_PERVOL;
AudioIO->ioa_Data                            =(BYTE *)waveptr;
AudioIO->ioa_Length                          =samples;
AudioIO->ioa_Period                          =clock*samcyc/(samples*frequency);
AudioIO->ioa_Volume                          =64;
AudioIO->ioa_Cycles                          =frequency*duration/samcyc;
printf("I/O block initialized to play tone...\n");
```

```
/*-----------------------------------------------------*/
/* Send the command to start a sound using BeginIO()    */
/* Go to sleep and wait for the sound to finish with    */
/* WaitPort().  When we wake-up we have to get the reply */
/*-----------------------------------------------------*/
printf("Starting tone now...\n");
BeginIO((struct IORequest *) AudioIO );
WaitPort(AudioMP);
AudioMSG = GetMsg(AudioMP);

printf("Sound finished...\n");

killaudio:

printf("Killing audio device...\n");
if (waveptr != 0)
    FreeMem(waveptr, 2);
if (device == 0)
    CloseDevice( (struct IORequest *) AudioIO );
if (AudioMP != 0)
    DeletePort(AudioMP);
if (AudioIO != 0)
    FreeMem( AudioIO,sizeof(struct IOAudio) );
}
```

Audio Allocation And Arbitration

The first command you send to the audio device should always be ADCMD_ALLOCATE. You can do this when you open the device, or at a later time. You specify the channels you want in the **ioa_Data** field of the **IOAudio** block. If the allocation succeeds, the audio device will return the channels that you now own in the lower four bits of the **ioa_Request.io_Unit** field of your **IOAudio** block. For instance, if the **io_Unit** field equals 5 (binary 0101) then you own channels 2 and 0. If the **io_Unit** field equals 15 (binary 1111) then you own all the channels.

When you send the ADCMD_ALLOCATE command, the audio device will also return a unique allocation key in the **ioa_AllocKey** of the **IOAudio** block. You must use this allocation key for all subsequent commands that you send to the audio device. The audio device uses this unique key to identify which task issued the command. If you do not use the correct allocation key assigned to you by the audio device when you send a command, your command will be ignored.

When you request a channel with ADCMD_ALLOCATE, you specify a precedence number from -128 to 127 in the **ioa_Request.io_Message.mn_Node.ln_Pri** field of the **IOAudio** block. If a channel you want is being used and you have specified a higher precedence than the current user, ADCMD_ALLOCATE will "steal" the channel from the other user. Later on, if your precedence is lower than that of another user who is performing an allocation, the channel may be stolen from you.

If you set the precedence to 127 when you open the device or raise the precedence to 127 with the ADCMD_SETPREC command, no other tasks can steal a channel from you. When you have finished with a channel, you must relinquish it with the ADCMD_FREE command to make it available for other users.

The following table shows suggested precedence values.

Suggested Precedences for Channel Allocation

Predecence	Type of Sound
127	*Unstoppable.* Sounds first allocated at lower precedence, then set to this highest level.
90 – 100	*Emergencies.* Alert, urgent situation that requires immediate action.
80 – 90	*Annunciators.* Attention, bell (CTRL-G).
75	*Speech.* Synthesized or recorded speech (narrator.device).
50 – 70	*Sonic cues.* Sounds that provide information that is not provided by graphics. Only the beginning of each sound (enough to recognize it) should be at this level; the rest should be set to sound effects level.
-50 – 50	*Music program.* Musical notes in music-oriented program. The higher levels should be used for the attack portions of each note.
-70 – -50	*Sound effects.* Sounds used in conjunction with graphics. More important sounds should use higher levels.
-100 – -80	*Background.* Theme music and restartable background sounds.
-128	*Silence.* Lowest level (freeing the channel completely is preferred).

If you attempt to perform a command on a channel that has been stolen from you by a higher priority task, an AUDIO_NOALLOCATION error is returned and the bit in the **ioa_Request.io_Unit** field corresponding to the stolen channel is cleared so you know which channel was stolen.

If you want to be warned before a channel is stolen so that you have a chance to stop your sound gracefully, then you should use the ADCMD_LOCK command after you open the device. This command is also useful for programs which write directly to the audio hardware. For more on ADCMD_LOCK, see the section below.

Allocation and Arbitration Commands

These commands allow the audio channels to be shared among different tasks and programs. None of these commands can be called from interrupt code.

ADCMD_ALLOCATE

This command gives your program a channel to use and should be the first command you send to the audio device. You specify the channels you want by setting a pointer to an array in the **ioa_Data** field of the **IOAudio** structure. This array uses a value of 1 to allocate channel 0, 2 for channel 1, 4 for channel 2, and 8 for channel 3. For multiple channels, add the values together. For example, if you want to allocate all channels, use a value of 15.

If you want a pair of stereo channels and you have no preference about which of the left and right channels the system will choose for the allocation, you can pass a pointer to an array containing 3, 5, 10, and 12. Channels 1 and 2 produce sound on the left side and channels 0 and 3 on the right side. The table below shows how this array corresponds to all the possible combinations of a right and a left channel.

Possible Channel Combinations

Channel 3 right	Channel 2 left	Channel 1 left	Channel 0 right	Decimal Value of Allocation Mask
0	0	1	1	3
0	1	0	1	5
1	0	1	0	10
1	1	0	0	12

How ADCMD_ALLOCATE Operates

The ADCMD_ALLOCATE command tries the first combination, 3, to see if channels 0 and 1 are not being used. If they are available, the 3 is copied into the **io_Unit** field and you get an allocation key for these channels in the **ioa_AllocKey** field. You copy the key into other I/O blocks for any other commands you may want to perform on these channels.

If channels 0 and 1 are being used, ADCMD_ALLOCATE tries the other combinations in turn. If all the combinations are in use, ADCMD_ALLOCATE checks the precedence number of the users of the channels and finds the combination that requires it to steal the channel or channels of the lowest precedence. If all the combinations require stealing a channel or channels of equal or higher precedence, the ADCMD_ALLOCATE request fails. Precedence is in the **ln_Pri** field of the **io_Message** in the **IOAudio** block you pass to ADCMD_ALLOCATE; it has a value from -128 to 127.

The ADIOF_NOWAIT Flag

If you need to produce a sound right now and otherwise don't want to allocate, set the ADIOF_NOWAIT flag to 1. This will cause the command to return an IOERR_ALLOCFAILED error if it cannot allocate any of the channels. If you are producing a non-urgent sound and you can wait, set the ADIOF_NOWAIT flag to 0. Then, the **IOAudio** block returns only when you get the allocation. If ADIOF_NOWAIT is set to 0, the audio device will continue to retry the allocation request whenever channels are freed until it is successful. If the program decides to cancel the request, **AbortIO()** can be used.

ADCMD_ALLOCATE Examples

The following are more examples of how to tell ADCMD_ALLOCATE your channel preferences. If you want any channel, but want a right channel first, use an array containing 1, 8, 2, and 4:

```
0001
1000
0010
0100
```

If you only want a right channel, use 1 and 8 (channels 0 and 3):

```
0001
1000
```

If you want only a left channel, use 2 and 4 (channels 1 and 2):

```
0010
0100
```

If you want to allocate a channel and keep it for a sound that can be interrupted and restarted, allocate it at a certain precedence. If it is stolen, allocate it again with the ADIOF_NOWAIT flag set to 0. When the channel is relinquished, you will get it again.

The Allocation Key

If you want to perform multi-channel commands, all the channels must have the same key since the **IOAudio** block has only one allocation key field. The channels must all have that same key even when they were not allocated simultaneously. If you want to use a key you already have, you can pass that key in the **ioa_AllocKey** field and ADCMD_ALLOCATE can allocate other channels with that existing key. The ADCMD_ALLOCATE command returns a new and unique key only if you pass it a zero in the allocation key field.

ADCMD_FREE

ADCMD_FREE is the opposite of ADCMD_ALLOCATE. When you perform ADCMD_FREE on a channel, it does a CMD_RESET command on the hardware and "unlocks" the channel. It also checks to see if there are other pending allocation requests. You do not need to perform ADCMD_FREE on channels stolen from you. If you want channels back after they have been stolen, you must reallocate them with the same allocation key.

ADCMD_SETPREC

This command changes the precedence of an allocated channel. As an example of the use of ADCMD_SETPREC, assume that you are making the sound of a chime that takes a long time to decay. It is important that user hears the chime but not so important that he hears it decay all the way. You could lower precedence after the initial attack portion of the sound to let another program steal the channel. You can also set the precedence to maximum (127) if you do not want the channel(s) stolen from you.

ADCMD_LOCK

The ADCMD_LOCK command performs the "steal verify" function. When another application is attempting to steal a channel or channels, ADCMD_LOCK gives you a chance to clean up before the channel is stolen. You perform a ADCMD_LOCK command right after the ADCMD_ALLOCATE command. ADCMD_LOCK does not return until a higher-priority user attempts to steal the channel(s) or you perform an ADCMD_FREE command. If someone is attempting to steal, you must finish up and ADCMD_FREE the channel as quickly as possible.

You must use ADCMD_LOCK if you want to write directly to the hardware registers instead of using the device commands. If your channel is stolen, you are not notified unless the ADCMD_LOCK command is present. This could cause problems for the task that has stolen the channel and is now using it at the same time as your task. ADCMD_LOCK sets a switch that is not cleared until you perform an ADCMD_FREE command on the channel. Canceling an ADCMD_LOCK request with **AbortIO()** will not free the channel.

The following outline describes how ADCMD_LOCK works when a channel is stolen and when it is not stolen.

1. User A allocates a channel.
2. User A locks the channel.

If User B allocates the channel with a higher precedence:

3. User B's ADCMD_ALLOCATE command is suspended (regardless of the setting of the ADIOF_NOWAIT flag).
4. User A's lock command is replied to with an error (ADIOERR_CHANNELSTOLEN).
5. User A does whatever is needed to finish up when a channel is stolen.
6. User A frees the channel with ADCMD_FREE.
7. User B's ADCMD_ALLOCATE command is replied to. Now user B has the channel.

If the channel is not allocated by another user:

3. User A finishes the sound.
4. User A performs the ADCMD_FREE command.
5. User A's ADCMD_LOCK command is replied to.

Never make the freeing of a channel (if the channel is stolen) dependent on allocating another channel. This may cause a deadlock. If you want channels back after they have been stolen, you must reallocate them with the same allocation key. To keep a channel and never let it be stolen, set precedence to maximum (127). Do not use a lock for this purpose.

Hardware Control Commands

The following commands change hardware registers and affect the actual sound output.

CMD_WRITE

This is a single-channel command and is the main command for making sounds. You pass the following to CMD_WRITE:

- A pointer to the waveform to be played (must start on a word boundary and must be in memory accessible by the custom chips, MEMF_CHIP)
- The length of the waveform in bytes (must be an even number)
- A count of how many times you want to play the waveform

If the count is 0, CMD_WRITE will play the waveform from beginning to end, then repeat the waveform continuously until something aborts it.

If you want period and volume to be set at the start of the sound, set the WRITE command's ADIOF_PERVOL flag. If you do not do this, the previous volume and period for that channel will be used. This is one of the flags that is cleared by **DoIO()** and **SendIO()**. The **ioa_WriteMsg** field in the **IOAudio** block is an extra message field that can be replied to at the start of the CMD_WRITE. This second message is used only to tell you when the CMD_WRITE command *starts* processing, and it is used only when the ADIOF_WRITEMESSAGE flag is set to 1.

If a CMD_STOP has been performed, the CMD_WRITE requests are queued up. The CMD_WRITE command does not make its own copy of the waveform, so any modification of the waveform before the CMD_WRITE command is finished may affect the sound. This is sometimes desirable for special effects. To splice together two waveforms without clicks or pops, you must send a separate, second CMD_WRITE command while the first is still in progress. This technique is used in double-buffering, which is described below.

By using two waveform buffers and two CMD_WRITE requests you can compute a waveform continuously. This is called double-buffering. The following describes how you use double-buffering.

1. Compute a waveform in memory buffer A.
2. Issue CMD_WRITE A with **io_Data** pointing to buffer A.
3. Continue the waveform in memory buffer B.
4. Issue CMD_WRITE B with **io_Data** pointing to Buffer B.
5. Wait for CMD_WRITE A to finish.
6. Continue the waveform in memory buffer A.
7. Issue CMD_WRITE A with **io_Data** pointing to Buffer A.
8. Wait for CMD_WRITE B to finish.
9. Loop back to step 3 until the waveform is finished.
10. At the end, remember to wait until both CMD_WRITE A and B are finished.

ADCMD_FINISH

The ADCMD_FINISH command aborts (calls **AbortIO()**) the current write request on a channel or channels. This is useful if you have something playing, such as a long buffer or some repetitions of a buffer, and you want to stop it.

ADCMD_FINISH has a flag you can set (ADIOF_SYNCCYCLE) that allows the waveform to finish the current cycle before aborting it. This is useful for splicing together sounds at zero crossings or some other place in the waveform where the amplitude at the end of one waveform matches the amplitude at the beginning of the next. Zero crossings are positions within the waveform at which the amplitude is zero. Splicing at zero crossings gives you fewer clicks and pops when the audio channel is turned off or the volume is changed.

ADCMD_PERVOL

ADCMD_PERVOL lets you change the volume and period of a CMD_WRITE that is in progress. The change can take place immediately or you can set the ADIOF_SYNCCYCLE flag to have the change occur at the end of the cycle. This is useful to produce vibratos, glissandos, tremolos, and volume envelopes in music or to change the volume of a sound.

CMD_FLUSH

CMD_FLUSH aborts (calls **AbortIO()**) all CMD_WRITEs and all ADCMD_WAITCYCLEs that are queued up for the channel or channels. It does not abort ADCMD_LOCKs (only ADCMD_FREE clears locks).

CMD_RESET

CMD_RESET restores all the audio hardware registers. It clears the attach bits, restores the audio interrupt vectors if the programmer has changed them, and performs the CMD_FLUSH command to cancel all requests to the channels. CMD_RESET also unstops channels that have had a CMD_STOP performed on them. CMD_RESET does not unlock channels that have been locked by ADCMD_LOCK.

ADCMD_WAITCYCLE

This is a single-channel command. ADCMD_WAITCYCLE is replied to when the current cycle has completed, If there is no CMD_WRITE in progress, it returns immediately.

CMD_STOP

This command stops the current write cycle immediately. If there are no CMD_WRITEs in progress, it sets a flag so any future CMD_WRITEs are queued up and do not begin processing (playing).

CMD_START

CMD_START undoes the CMD_STOP command. Any cycles that were stopped by the CMD_STOP command are actually lost because of the impossibility of determining exactly where the DMA ceased. If the CMD_WRITE command was playing two cycles and the first one was playing when CMD_STOP was issued, the first one is lost and the second one will be played.

This command is also useful when you are playing the same wave form with the same period out of multiple channels. If the channels are stopped when the CMD_WRITE commands are issued, CMD_START exactly synchronizes them, avoiding cancellation and distortion. When channels are allocated, they are effectively started by the CMD_START command.

CMD_READ

CMD_READ is a single-channel command. Its only function is to return a pointer to the current CMD_WRITE command. It enables you to determine which request is being processed.

Double Buffered Sound Example

The program listed below demonstrates double buffering with the audio device. Run the program from the CLI. It takes one parameter—the name of an IFF 8SVX sample file to play on the Amiga's audio device. The maximum size for a sample on the Amiga is 128K. However, by using double-buffering and queueing up requests to the audio device, you can play longer samples smoothly and without breaks.

```
/*
 * Audio_8SVX.c
 *
 * 8SVX example - double buffers >128K samples
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <devices/audio.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <graphics/gfxbase.h>
#include <iff/iff.h>
#include <iff/8svx.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>
#include <clib/graphics_protos.h>

#include <stdlib.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

#define VHDR MakeID('V','H','D','R')
#define BODY MakeID('B','O','D','Y')
#define MY8S MakeID('8','S','V','X')

void              kill8svx(char *);
void              kill8(void);

/*--------------------*/                /* These globals are needed */
/*   G L O B A L S   */                 /* by the clean up routines */
/*--------------------*/
struct IOAudio      *AIOptr1,           /* Pointers to Audio IOBs      */
                    *AIOptr2,
                    *Aptr;
struct Message      *msg;               /* Msg, port and device for    */
struct MsgPort      *port,              /* driving audio               */
                    *port1,*port2;
       ULONG         device;
       UBYTE        *sbase,*fbase;      /* For sample memory allocation */
       ULONG         fsize,ssize;       /* and freeing                  */

struct FileHandle   *v8handle;
       UBYTE         chan1[]  = {  1 }; /* Audio channel allocation arrays */
       UBYTE         chan2[]  = {  2 };
       UBYTE         chan3[]  = {  4 };
       UBYTE         chan4[]  = {  8 };
       UBYTE        *chans[] = {chan1,chan2,chan3,chan4};

       BYTE          oldpri,c;          /* Stuff for bumping priority */

struct Task         *mt=0L;
struct GfxBase      *GfxBase = NULL;
```

```
/*-----------*/
/*  M A I N  */
/*-----------*/

void main(int argc,char **argv)
{

/*-------------*/
/* L O C A L S */
/*-------------*/

       char          *fname;              /* File name and data pointer*/
       UBYTE         *p8data;             /* for file read.           */
       ULONG          clock;              /* Clock constant           */
       ULONG          length[2];          /* Sample lengths           */
       BYTE           iobuffer[8],        /* Buffer for 8SVX header   */
                     *psample[2];         /* Sample pointers          */
       Chunk         *p8Chunk;            /* Pointers for 8SVX parsing */
       Voice8Header *pVoice8Header;
       ULONG          y,rd8count,speed;   /* Counters, sampling speed   */
       ULONG          wakebit;            /* A wakeup mask              */

/*-------------*/
/*   C O D E   */
/*-------------*/

/*------------------------------*/
/* Check Arguments, Initialize  */
/*------------------------------*/

fbase=0L;
sbase=0L;
AIOptr1=0L;
AIOptr2=0L;
port=0L;
port1=0L;
port2=0L;
v8handle=0L;
device=1L;

if (argc < 2)
    {
    kill8svx("No file name given.\n");
    exit(1L);
    }
fname=argv[1];

/*---------------------------*/
/* Initialize Clock Constant */
/*---------------------------*/

GfxBase=(struct GfxBase *)OpenLibrary("graphics.library",0L);
if (GfxBase==0L)
    {
    puts("Can't open graphics library\n");
    exit(1L);
    }

if (GfxBase->DisplayFlags & PAL)
    clock=3546895L;        /* PAL clock */
else
    clock=3579545L;        /* NTSC clock */

if (GfxBase)
    CloseLibrary( (struct Library *) GfxBase);

/*---------------*/
/* Open the File */
/*---------------*/

v8handle= (struct FileHandle *) Open(fname,MODE_OLDFILE);
if (v8handle==0)
    {
    kill8svx("Can't open 8SVX file.\n");
    exit(1L);
    }
```

```
/*-------------------------------------------*/
/* Read the 1st 8 Bytes of the File for Size */
/*-------------------------------------------*/
rd8count=Read((BPTR)v8handle,iobuffer,8L);
if (rd8count==-1)
    {
    kill8svx ("Read error.\n");
    exit(1L);
    }

if (rd8count<8)
    {
    kill8svx ("Not an IFF 8SVX file, too short\n");
    exit(1L);
    }

/*-----------------*/
/* Evaluate Header */
/*-----------------*/
p8Chunk=(Chunk *)iobuffer;
if (p8Chunk->ckID != FORM )
    {
    kill8svx("Not an IFF FORM.\n");
    exit(1L);
    }

/*--------------------------------------------*/
/* Allocate Memory for File and Read it in.   */
/*--------------------------------------------*/
fbase= (UBYTE *)AllocMem(fsize=p8Chunk->ckSize , MEMF_PUBLIC|MEMF_CLEAR);
if (fbase==0)
    {
    kill8svx("No memory for read.\n");
    exit(1L);
    }

p8data=fbase;

rd8count=Read((BPTR)v8handle,p8data,p8Chunk->ckSize);
if (rd8count==-1)
    {
    kill8svx ("Read error.\n");
    exit(1L);
    }

if (rd8count<p8Chunk->ckSize)
    {
    kill8svx ("Malformed IFF, too short.\n");
    exit(1L);
    }

/*-------------------*/
/* Evaluate IFF Type */
/*-------------------*/
if (MakeID( *p8data, *(p8data+1) , *(p8data+2) , *(p8data+3) ) != MY8S )
    {
    kill8svx("Not an IFF 8SVX file.\n");
    exit(1L);
    }

/*----------------------*/
/* Evaluate 8SVX Chunks */
/*----------------------*/
p8data=p8data+4;

while( p8data < fbase+fsize )
  {
  p8Chunk=(Chunk *)p8data;

  switch(p8Chunk->ckID)
    {
    case VHDR:
      /*---------------------------------------------*/
      /* Get a pointer to the 8SVX header for later use */
      /*---------------------------------------------*/
      pVoice8Header=(Voice8Header *)(p8data+8L);
      break;
    case BODY:
      /*-----------------------------------------------*/
      /* Create pointers to 1-shot and continuous parts  */
      /* for the top octave and get length. Store them.  */
      /*-----------------------------------------------*/
        psample[0] = (BYTE *)(p8data + 8L);
        psample[1] = psample[0] + pVoice8Header->oneShotHiSamples;
        length[0] = (ULONG)pVoice8Header->oneShotHiSamples;
        length[1] = (ULONG)pVoice8Header->repeatHiSamples;
        break;

    default:
      break;
    }
    /* end switch */

  p8data = p8data + 8L + p8Chunk->ckSize;

  if (p8Chunk->ckSize&1L == 1)
      p8data++;
  }

/* Play either the one-shot or continuous, not both */
if (length[0]==0)
    y=1;
else
    y=0;

/*-------------------------------------*/
/* Allocate chip memory for samples and  */
/* copy from read buffer to chip memory. */
/*-------------------------------------*/
if (length[y]<=102400)
    ssize=length[y];
else
    ssize=102400;

sbase=(UBYTE *)AllocMem( ssize , MEMF_CHIP | MEMF_CLEAR);
if (sbase==0)
    {
    kill8svx("No chip memory.\n");
    exit(1L);
    }

CopyMem(psample[y],sbase,ssize);
psample[y]+=ssize;

/*--------------------------------*/
/* Calculate playback sampling rate */
/*--------------------------------*/
speed =  clock / pVoice8Header->samplesPerSec;

/*------------------*/
/* Bump our priority */
/*------------------*/
mt=FindTask(NULL);
oldpri=SetTaskPri(mt,21);

/*-------------------------------*/
/* Allocate two audio I/O blocks  */
/*-------------------------------*/
AIOptr1=(struct IOAudio *)
      AllocMem( sizeof(struct IOAudio),MEMF_PUBLIC|MEMF_CLEAR);
if (AIOptr1==0)
    {
    kill8svx("No IO memory\n");
    exit(1L);
    }

AIOptr2=(struct IOAudio *)
      AllocMem( sizeof(struct IOAudio),MEMF_PUBLIC|MEMF_CLEAR);
if (AIOptr2==0)
    {
    kill8svx("No IO memory\n");
    exit(1L);
    }
```

```
/*----------------------*/
/* Make two reply ports */
/*----------------------*/

port1=CreatePort(0,0);
if (port1==0)
    {
    kill8svx("No port\n");
    exit(1L);
    }

port2=CreatePort(0,0);
if (port2==0)
    {
    kill8svx("No port\n");
    exit(1L);
    }

c=0;
while(device!=0 && c<4)
  {
  /*---------------------------------------*/
  /* Set up audio I/O block for channel    */
  /* allocation and Open the audio device  */
  /*---------------------------------------*/
  AIOptr1->ioa_Request.io_Message.mn_ReplyPort    = port1;
  AIOptr1->ioa_Request.io_Message.mn_Node.ln_Pri = 127;  /* No stealing! */
  AIOptr1->ioa_AllocKey                          = 0;
  AIOptr1->ioa_Data                              = chans[c];
  AIOptr1->ioa_Length                            = 1;

  device=OpenDevice(AUDIONAME,0L,(struct IORequest *)AIOptr1,0L);
  c++;
  }

if (device!=0)
    {
    kill8svx("No channel\n");
    exit(1L);
    }

/*-------------------------------------------*/
/* Set Up Audio IO Blocks for Sample Playing */
/*-------------------------------------------*/

AIOptr1->ioa_Request.io_Command   =CMD_WRITE;
AIOptr1->ioa_Request.io_Flags     =ADIOF_PERVOL;

/*--------*/
/* Volume */
/*--------*/

AIOptr1->ioa_Volume=60;

/*---------------*/
/* Period/Cycles */
/*---------------*/

AIOptr1->ioa_Period =(UWORD)speed;
AIOptr1->ioa_Cycles =1;

*AIOptr2 = *AIOptr1;    /* Make sure we have the same allocation keys, */
                        /* same channels selected and same flags       */
                        /* (but different ports...)                    */

AIOptr1->ioa_Request.io_Message.mn_ReplyPort    = port1;
AIOptr2->ioa_Request.io_Message.mn_ReplyPort    = port2;

/*--------*/
/*  Data  */
/*--------*/

AIOptr1->ioa_Data             =(UBYTE *)sbase;
AIOptr2->ioa_Data             =(UBYTE *)sbase + 51200;
```

```
/*-----------------*/
/*  Run the sample */
/*-----------------*/

if (length[y]<=102400)
    {
    AIOptr1->ioa_Length=length[y];          /* No double buffering needed */
    BeginIO((struct IORequest *)AIOptr1);  /* Begin the sample, wait for */
    wakebit=0L;                             /* it to finish, then quit.   */
    wakebit=Wait(1 << port1->mp_SigBit);
    while((msg=GetMsg(port1))==0){};
    }
else
    {
    length[y]-=102400;                     /* It's a real long sample so  */
    AIOptr1->ioa_Length=51200L;             /* double buffering is needed  */
    AIOptr2->ioa_Length=51200L;
    BeginIO((struct IORequest *)AIOptr1); /* Start up the first 2 blocks... */
    BeginIO((struct IORequest *)AIOptr2);
    Aptr=AIOptr1;
    port=port1;                            /* Set the switch... */

    while(length[y]>0)
      {                                    /* We Wait() for one IO to finish, */
      wakebit=Wait(1 << port->mp_SigBit); /* then reuse the IO block & queue */
      while((msg=GetMsg(port))==0){};      /* it up again while the 2nd IO    */
                                           /* block plays. Switch and repeat. */
      /* Set length of next IO block */
      if (length[y]<=51200)
          Aptr->ioa_Length=length[y];
      else
          Aptr->ioa_Length=51200L;

      /* Copy sample fragment from read buffer to chip memory */
      CopyMem(psample[y],Aptr->ioa_Data,Aptr->ioa_Length);

      /* Adjust size and pointer of read buffer*/
      length[y]-=Aptr->ioa_Length;
      psample[y]+=51200;

      BeginIO((struct IORequest *)Aptr);

      if (Aptr==AIOptr1)
          {
          Aptr=AIOptr2;                   /* This logic handles switching  */
          port=port2;                     /* between the 2 IO blocks and   */
          }                                /* the 2 ports we are using.     */
      else
          {
          Aptr=AIOptr1;
          port=port1;
          }
      }

    /*-------------------------------------------------*/
    /* OK we are at the end of the sample so just wait */
    /* for the last two parts of the sample to finish  */
    /*-------------------------------------------------*/
    wakebit=Wait(1 << port->mp_SigBit);
    while((msg=GetMsg(port))==0){};
    if (Aptr==AIOptr1)
        {
        Aptr=AIOptr2;                   /* This logic handles switching  */
        port=port2;                     /* between the 2 IO blocks and   */
        }                               /* the 2 ports we are using.     */
    else
        {
        Aptr=AIOptr1;
        port=port1;
        }
    wakebit=Wait(1 << port->mp_SigBit);
    while((msg=GetMsg(port))==0){};
    }

kill8();
exit(0L);
}
```

```
/*-----------------*/
/* Abort the Read */
/*-----------------*/
void
kill8svx(kill8svxstring)
char *kill8svxstring;
{
puts(kill8svxstring);
kill8();
}

/*-------------------------*/
/* Return system resources */
/*-------------------------*/
void
kill8()
{
if (device ==0)
    CloseDevice((struct IORequest *)AIOptr1);
if (port1  !=0)
    DeletePort(port1);
if (port2  !=0)
    DeletePort(port2);
if (AIOptr1!=0)
    FreeMem( AIOptr1,sizeof(struct IOAudio) );
if (AIOptr2!=0)
    FreeMem( AIOptr2,sizeof(struct IOAudio) );

if (mt!=0)
    SetTaskPri(mt,oldpri);

if (sbase !=0)
    FreeMem (sbase, ssize);
if (fbase !=0)
    FreeMem(fbase,fsize);
if (v8handle!=0)
    Close((BPTR)v8handle);
}
```

Additional Information on the Audio Device

Additional programming information on the audio device can be found in the include files and the Autodocs for the audio device. Both are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*. Information can also be found in the *Amiga Hardware Reference Manual.*

Audio Device Information	
INCLUDES	devices/audio.h devices/audio.i
AUTODOCS	audio.doc

chapter three
CLIPBOARD DEVICE

The clipboard device allows the exchange of data dynamically between one application and another. It is responsible for caching data that has been "cut" and providing data to "paste" in an application. A special "post" mode allows an application to inform the clipboard device that the application has data available. The clipboard device will request this data only if the data is actually needed. The clipboard will cache the data in RAM and will automatically spool the data to disk if necessary.

The clipboard device is implemented as an Exec-style device, and supports random access reads and writes on data within the clipboard. All data in the clipboard must be in IFF format. A new library, iffparse.library, has been added to the Amiga libraries. The routines in iffparse.library can and should be used for reading and writing data to the clipboard. This chapter contains a brief discussion of IFF as it relates to the clipboard (for more details see Appendix A).

New Clipboard Features for Version 2.0

Feature	Description
CBD_CHANGEHOOK	Device Command

Compatibility Warning: The new features for the 2.0 clipboard device are not backwards compatible.

Clipboard Device Commands and Functions

Command	Command Operation
CBD_CHANGEHOOK	Specify a hook to be called when the data on the clipboard has changed (V36).
CBD_CURRENTREADID	Return the Clip ID of the current clip to read. This is used to determine if a clip posting is still the latest cut.
CBD_CURRENTWRITEID	Return the Clip ID of the latest clip written. This is used to determine if the clip posting data is obsolete.
CBD_POST	Post the availability of clip data.
CMD_READ	Read data from the clipboard for a paste. Data can be read from anywhere in the clipboard by specifying an offset >0 in the I/O request.
CMD_UPDATE	Indicate that the data provided with a write command is complete and available for subsequent read/pastes.
CMD_WRITE	Write data to the clipboard as a cut.

Exec Functions as Used in This Chapter

CloseDevice()	Relinquish use of the clipboard device. All requests must be complete before closing.
DoIO()	Initiate a command and wait for completion (synchronous request).
GetMsg()	Get next message from a message port.
OpenDevice()	Obtain use of the clipboard device.
SendIO()	Initiate a command and return immediately (asynchronous request).

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an I/O request structure of type **IOClipReq**. This structure will be used to communicate commands to the clipboard device.
CreatePort()	Create a signal message port for reply messages from the clipboard device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete an I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The clipboard device operates like the other Amiga devices. To use it, you must first open the clipboard device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

```
struct IOClipReq
{
    struct  Message io_Message;
    struct  Device  *io_Device;     /* device node pointer  */
    struct  Unit    *io_Unit;       /* unit (driver private)*/
    UWORD   io_Command;             /* device command */
    UBYTE   io_Flags;               /* including QUICK and SATISFY */
    BYTE    io_Error;               /* error or warning num */
    ULONG   io_Actual;              /* number of bytes transferred */
    ULONG   io_Length;              /* number of bytes requested */
    STRPTR  io_Data;                /* either clip stream or post port */
    ULONG   io_Offset;              /* offset in clip stream */
    LONG    io_ClipID;              /* ordinal clip identifier */
};
```

See the include file *devices/clipboard.h* for the complete structure definition.

The clipboard device I/O request, **IOClipReq**, looks like a standard **IORequest** structure except for the addition of the **io_ClipID** field, which is used by the device to identify clips. It must be set to zero by the application for a post or an initial write or read, but preserved for subsequent writes or reads, as the clipboard device uses this field internally for bookkeeping purposes.

OPENING THE CLIPBOARD DEVICE

Three primary steps are required to open the clipboard device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an extended I/O request structure of type **IOClipReq** using **CreateExtIO()**.
- Open the clipboard device. Call **OpenDevice()**, passing the **IOClipReq**.

```
struct MsgPort   *ClipMP;          /* pointer to message port*/
struct IOClipReq *ClipIO;          /* pointer to IORequest */

if (ClipMP=CreatePort(0L,0L) )
    {
    if (ClipIO=(struct IOClipReq *)
                CreateExtIO(ClipMP,sizeof(struct IOClipReq)))
        {
        if (OpenDevice("clipboard.device",0L,ClipIO,0))
            printf("clipboard.device did not open\n");
        else
            {
             ... do device processing
            }
        {
    else
        printf("Error: Could not create IORequest\n");
    }
else
    printf("Error: Could not create message port\n");
```

CLIPBOARD DATA

Data on the clipboard resides in one of three places. When an application posts a cut, the data resides in the private memory space of that application. When an application writes to the clipboard, either of its own volition or in response to a message from the clipboard requesting that it satisfy a post, the data is copied to the clipboard which is either memory or a special disk file. When the clipboard is not open, the data resides in the special disk file located in the directory specified by the CLIPS: logical AmigaDOS assign.

Data on the clipboard is self-identifying. It must be a correct IFF (Interchange File Format) file; the rest of this section refers to IFF concepts. See the Appendix A in this manual for a complete description of IFF. If the top-level chunk is of type CAT with an identifier of CLIP, that indicates that the contained chunks are different representations of the same data, in decreasing order of preference on the part of the producer of the clip. Any other data is as defined elsewhere (probably a single representation of the cut data produced by an application).

The iffparse.library also contains functions which simplify reading and writing of IFF data to the clipboard device. See the "IFF Parse Library" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

A clipboard tool, which is an application that allows a Workbench user to view a clip, should understand the text (FTXT) and graphics (ILBM) form types. Applications using the clipboard to export data should include at least one of these types in a CAT CLIP so that their data can be represented on the clipboard in some form for user feedback.

You should not, in any way, rely on the specifics of how files in CLIPS: are handled or named. The only proper way to read or write clipboard data is via the clipboard device.

> *Play Nice!* Keep in mind that while your task is reading from or writing to a clipboard unit, other tasks cannot. Therefore, it is important to be fast. If possible, make a copy of the clipboard data in RAM instead of processing it while the read or write is in progress.

MULTIPLE CLIPS

The clipboard supports multiple clips, i.e., the clipboard device can contain more than one distinct piece of data. This is not to be confused with the IFF CAT CLIP, which allows for different representation of the same data.

The multiple clips are implemented as different units in the clipboard device. The unit is specified at **OpenDevice()** time.

```
struct IOClipReq *ClipIO;
LONG unit;

OpenDevice("clipboard.device", unit, ClipIO, 0);
```

By default, applications should use clipboard unit 0. However, it is recommended that each application provide a mechanism for selecting the unit number which will be used when the clipboard is opened. This will allow the user to create a convention for storing different types of data in the clipboard. Applications should never write to clipboard unit 0 unless the user requests it (e.g., selecting COPY or CUT within an application).

Clipboard units 1–255 can be used by the more advanced user for:

- Sharing data between applications within an ARexx Script.
- Customizing applications to store different kinds of data in different clipboard units.
- Customizing an application to use multiple cut/copy/paste buffers.
- Specialized utilities which might display and/or automatically modify the contents of a clipboard unit.

All applications which provide CUT, COPY and PASTE capabilities, should, at a minimum, provide support for clipboard unit 0.

WRITING TO THE CLIPBOARD DEVICE

You write to the clipboard device by passing an **IOClipReq** to the device with **CMD_WRITE** set in **io_Command**, the number of bytes to be written set in **io_Length** and the address of the write buffer set in **io_Data**.

```
ClipIO->io_Data = (char *) data;
ClipIO->io_Length = 4L;
ClipIO->io_Command = CMD_WRITE;
```

An initial write should set **io_Offset** to zero. Each time a write is done, the device will increment **io_Offset** by the length of the write.

As previously stated, the data you write to the clipboard must be in IFF format. This requires a certain amount of preparation prior to actually writing the data if it is not already in IFF format. A brief explanation of the IFF format will be helpful in this regard.

For our purposes, we will limit our discussion to a simple formatted text (FTXT) IFF file. An FTXT file looks like:

FORM
length of succeeding bytes
FTXT
CHRS
length of succeeding bytes
data bytes
pad byte of zero if the preceding chunk has odd length

Based on the above figure, a hex dump of an IFF FTXT file containing the string Enterprise would look like:

0000	464F524D	FORM
0004	00000016	(length of FTXT, CHRS, 0xA and data)
0008	46545854	FTXT
000C	43485253	CHRS
0010	0000000A	(length of Enterprise)
0014	456E7465	Ente
0018	72707269	rpri
001C	7365	se

A code fragment for doing this:

```
LONG slen = strlen ("Enterprise");
BOOL odd = (slen & 1);      /* pad byte flag */

/* set length depending on whether string is odd or even length */
LONG length = (odd) ? slen + 1 : slen;

/* Reset the clip id */
ClipIO->io_ClipID = 0;
ClipIO->io_Offset = 0;

error = writeLong ((LONG *) "FORM");/* "FORM" */

length += 12;  /* add 12 bytes for FTXT, CHRS & length byte to string length */
error = writeLong (&length);
error = writeLong ((LONG *) "FTXT");/* "FTXT" for example */
error = writeLong ((LONG *) "CHRS");/* "CHRS" for example */
error = writeLong (&slen);      /* #  (length of string) */

ClipIO->io_Command = CMD_WRITE;
ClipIO->io_Data = (char *) string;
ClipIO->io_Length = slen;                /* length of string */
error = (LONG) DoIO (clipIO);    /* text string */


LONG writeLong (LONG * ldata)
{
    ClipIO->io_Command = CMD_WRITE;
    ClipIO->io_Data = (char *) ldata;
    ClipIO->io_Length = 4L;
    return ( (LONG) DoIO (clipIO) );
}
```

The fragment above does no error checking because it's a fragment. You should always error check. See the example programs at the end of this chapter for the proper method of error checking.

> *Iffparse That Data!* Keep in mind that the functions in the iffparse.library can be used to write data to the clipboard. See the "IFF Parse Library" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

UPDATING THE CLIPBOARD DEVICE

When the final write is done, an update command must be sent to the device to indicate that the writing is complete and the data is available. You update the clipboard device by passing an **IOClipReq** to the device with CMD_UPDATE set in **io_Command**.

```
ClipIO->io_Command = CMD_UPDATE;
DoIO(ClipIO);
```

CLIPBOARD MESSAGES

When an application performs a post, it must specify a message port for the clipboard to send a message to if it needs the application to satisfy the post with a write called the **SatisfyMsg**.

```
struct SatisfyMsg
{
struct  Message sm_Message; /* the length will be 6 */
UWORD   sm_Unit;            /* 0 for the primary clip unit */
LONG    sm_ClipID;          /* the clip identifier of the post */
}
```

This structure is defined in the include file *devices/clipboard.h.*

If the application wishes to determine if a post it has recently performed is still the current clip, it should compare the **io_ClipID** found in the post request upon return with that returned by the CBD_CURRENTREADID command.

If an application has a pending post and wishes to determine if it should satisfy it (for example, before it exits), it should compare the **io_ClipID** of the post I/O request with that of the CBD_CURRENTWRITEID command. If the application receives a satisfy message from the clipboard device (format described below), it must immediately perform the write with the **io_ClipID** of the post. The satisfy message from the clipboard may be removed from the application message port by the clipboard device at any time (because it is re-used by the clipboard device). It is not dangerous to spuriously satisfy a post, however, because it is identified by the **io_ClipID.**

The cut data is provided to the clipboard device via either a write or a post of the cut data. The write command accepts the data immediately and copies it onto the clipboard. The post command allows an application to inform the clipboard of a cut, but defers the write until the data is actually required for a paste.

In the preceding discussion, references to the read and write commands of the clipboard device actually refer to a sequence of read or write commands, where the clip data is acquired and provided in pieces instead of all at once.

The clipboard has an end-of-clip concept that is analogous to end-of-file for both read and write. The read end-of-file must be triggered by the user of the clipboard in order for the clipboard to move on to service another application's requests, and consists of reading data past the end of file. The write end-of-file is indicated by use of the update command, which indicates to the clipboard that the previous write commands are completed.

READING FROM THE CLIPBOARD DEVICE

You read from the clipboard device by passing an **IOClipReq** to the device with CMD_READ set in **io_Command,** the number of bytes to be read set in **io_Length** and the address of the read buffer set in **io_Data.**

```
ClipIO->io_Command = CMD_READ;
ClipIO->io_Data = (char *) read_data;
ClipIO->io_Length = 20L;
```

io_Offset must be set to zero for the first read of a paste sequence. An **io_Actual** that is less than the **io_Length** indicates that all the data has been read. After all the data has been read, a subsequent read must be performed (one whose **io_Actual** returns zero) to indicate to the clipboard device that all the data has been read. This allows random access of the clip while reading. Providing only valid reads are performed, your program can seek/read anywhere within the clip by setting the **io_Offset** field of the I/O request appropriately.

> *Tell The Clipboard You Are Finished Reading.* Your application must perform an extra read (one whose **io_Actual** returns zero) to indicate to the clipboard device that all data has been read, *if* **io_Actual** *is not already zero.*

The data you read from the clipboard will be in IFF format. Conversion from IFF may be necessary depending on your application.

Iffparse That Data! Keep in mind that the functions in the iffparse.library can be used to read data from the clipboard. See the "IFF Parse Library" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

CLOSING THE CLIPBOARD DEVICE

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**.

```
CloseDevice(ClipIO);
```

When the last task closes a clipboard unit with **CloseDevice()**, the contents of the unit may be copied to a disk file in CLIPS: so that the clipboard device can be expunged.

Monitoring Clipboard Changes

Some applications require notification of changes to data on the clipboard. Typically, these applications will need to do some processing when this occurs. You can set up such an environment through the CBD_CHANGEHOOK command. CBD_CHANGEHOOK allows you to specify a hook to be called when the data on the clipboard changes.

For example, a show clipboard utility would need to know when the data on the clipboard is changed so that it can display the new data. The hook it would specify would read the new clipboard data and display it for the user.

You specify a hook for the clipboard device by initializing a **Hook** structure and then passing an **IOClipReq** to the device with CBD_CHANGEHOOK set in **io_Command**, 1 set in **io_Length**, and the address of the **Hook** structure set in **io_Data**.

```
ULONG HookEntry ();                /* Declare the hook assembly function */
struct IOClipReq *ClipIO;          /* Declare the IOClipReq */
struct Hook *ClipHook;             /* Declare the Hook */

/* Prepare the hook */
ClipHook->h_Entry = HookEntry;          /* C interface in assembly routine HookEntry */
ClipHook->h_SubEntry = HookFunc;        /* Function to call when Hook is activated */
ClipHook->h_Data = FindTask(NULL);      /* Set pointer to current task */

ClipIO->io_Data = (char *) ClipHook;      /* Point to hook struct */
ClipIO->io_Length = 1;                    /* Add hook to clipboard */
ClipIO->io_Command = CBD_CHANGEHOOK;
DoIO(clipIO);
```

The above code fragment assumes that an assembly language routine **HookEntry()** has been coded:

```
; entry interface for C code
_HookEntry:
        move.l  a1,-(sp)              ; push message packet pointer
        move.l  a2,-(sp)              ; push object pointer
        move.l  a0,-(sp)              ; push hook pointer
        move.l  h_SubEntry(a0),a0     ; fetch C entry point ...
        jsr     (a0)                  ; ... and call it
        lea     12(sp),sp             ; fix stack
        rts
```

It also assumes that the function **HookFunc()** has been coded. One of the example programs at the end of this chapter has hook processing in it. See the include file *utility/hooks.h* and *The Amiga ROM Kernel Reference Manual: Libraries* for further information on hooks.

You remove a hook by passing an **IOClipReq** to the device with the address of the **Hook** structure set in **io_Data**, 0 set in **io_Length** and CBD_CHANGEHOOK set in **io_Command**.

```
ClipIO->io_Data = (char *) ClipHook;     /* point to hook struct */
ClipIO->io_Length = 0;                   /* Remove hook from clipboard */
ClipIO->io_Command = CBD_CHANGEHOOK;
(DoIO (clipIO))
```

You must remove the hook or it will continue indefinitely.

CAVEATS FOR CBD_CHANGEHOOK

- CBD_CHANGEHOOK should only be used by a special application, such as a clipboard viewing program. Most applications can check the contents of the clipboard when, and if, the user requests a paste.
- Do not add system overhead by blindly reading and parsing the clipboard everytime a user copies data to it. If all applications did this, the system could become intolerably slow whenever an application wrote to the clipboard. Only read and parse when it is necessary.

Example Clipboard Programs

```
/*
 * Clipdemo.c
 *
 * Demonstrate use of clipboard I/O.  Uses general functions
 * provided in cbio.c
 *
 * Compile with SAS C 5.10: LC -b1 -cfistq -v -y -L+cbio.o
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/ports.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <devices/clipboard.h>
#include <libraries/dosextens.h>
#include <libraries/dos.h>

#include "cb.h"

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

#define FORGETIT 0
#define READIT   1
#define WRITEIT  2
```

```
#define POSTIT   3

/* prototypes */

int ReadClip( void );           /* Demonstrate reading clipboard data      */
int WriteClip( char * );        /* Demonstrate write to clipboard         */
int PostClip( char * );         /* Demonstrate posting data to clipboard */

void main( USHORT, char **);

char message[] = "\
\nPossible switches are:\n\n\
-r            Read, and output contents of clipboard.\n\
-w [string]   Write string to clipboard.\n\n\
-p [string]   Write string to clipboard using the clipboard POST mechanism.\n\n\
              The Post can be satisfied by reading data from\n\
              the clipboard.  Note that the message may never\n\
              be received if some other application posts, or\n\
              performs an immediate write to the clipboard.\n\n\
              To run this test you must run two copies of this example.\n\
              Use the -p switch with one to post data, and the -r switch\n\
              with another to read the data.\n\n\
              The process can be stopped by using the BREAK command,\n\
              in which case this example checks the CLIP write ID\n\
              to determine if it should write to the clipboard before\n\
              exiting.\n\n";


void main(argc,argv)
USHORT argc;
char **argv;
{

int todo;
char *string;

todo = FORGETIT;

if (argc)     /* from CLI ? */
    {

    /* Very simple code to parse for arguments - will suffice for
     * the sake of this example
     */

    if (argc > 1)
       {
       if (!(strcmp(argv[1],"-r")))
           todo = READIT;
       if (!(strcmp(argv[1],"-w")))
           todo = WRITEIT;
       if (!(strcmp(argv[1],"-p")))
           todo = POSTIT;

       string = NULL;

       if (argc > 2)
           string=argv[2];

       }

    switch (todo)
           {

           case READIT:

                ReadClip();
                break;

           case POSTIT:

                PostClip(string);
                break;

           case WRITEIT:

                WriteClip(string);
```

```
                  break;

             default:

                  printf("%s",message);
                  break;

             }


    }
}

/*
 * Read, and output FTXT in the clipboard.
 *
 */

ReadClip()
{
struct IOClipReq *ior;
struct cbbuf *buf;


/* Open clipboard.device unit 0 */

if (ior=CBOpen(0L))
    {

    /* Look for FTXT in clipboard */

    if (CBQueryFTXT(ior))
        {

        /* Obtain a copy of the contents of each CHRS chunk */

        while (buf=CBReadCHRS(ior))
              {
              /* Process data */

              printf("%s\n",buf->mem);

              /* Free buffer allocated by CBReadCHRS() */

              CBFreeBuf(buf);
              }

        /* The next call is not really needed if you are sure */
        /* you read to the end of the clip.                    */

        CBReadDone(ior);
        }
    else
        {
        puts("No FTXT in clipboard");
        }

    CBClose(ior);
    }

else
    {
    puts("Error opening clipboard unit 0");
    }

return(0L);
}

/*
 * Write a string to the clipboard
 *
 */

WriteClip(string)
char *string;
{
```

```
struct IOClipReq *ior;

if (string == NULL)
    {
    puts("No string argument given");
    return(0L);
    }

/* Open clipboard.device unit 0 */

if (ior = CBOpen(0L))
    {
    if (!(CBWriteFTXT(ior,string)))
        {
        printf("Error writing to clipboard: io_Error = %ld\n",ior->io_Error);
        }
    CBClose(ior);
    }
else
    {
    puts("Error opening clipboard.device");
    }

return(0);
}


/*
 * Write a string to the clipboard using the POST mechanism
 *
 * The POST mechanism can be used by applications which want to
 * defer writing text to the clipboard until another application
 * needs it (by attempting to read it via CMD_READ).  However
 * note that you still need to keep a copy of the data until you
 * receive a SatisfyMsg from the clipboard.device, or your program
 * exits.
 *
 * In most cases it is easier to write the data immediately.
 *
 * If your program receives the SatisfyMsg from the clipboard.device,
 * you MUST write some data.  This is also how you reply to the message.
 *
 * If your program wants to exit before it has received the SatisfyMsg,
 * you must check the io_ClipID field at the time of the post against
 * the current post ID which is obtained by sending the CBD_CURRENTWRITEID
 * command.
 *
 * If the value in io_ClipID (returned by CBD_CURRENTWRITEID) is greater
 * than your post ID, it means that some other application has performed
 * a post, or immediate write after your post, and that you're application
 * will never receive the SatisfyMsg.
 *
 * If the value in io_ClipID (returned by CBD_CURRENTWRITEID) is equal
 * to your post ID, then you must write your data, and send CMD_UPDATE
 * before exiting.
 *
 */

PostClip(string)
char *string;
{

struct MsgPort *satisfy;
struct SatisfyMsg *sm;
struct IOClipReq *ior;
int mustwrite;
ULONG postID;

if (string == NULL)
    {
    puts("No string argument given");
    return(0L);
    }

if (satisfy = CreatePort(0L,0L))
    {
```

```
    /* Open clipboard.device unit 0 */

    if (ior = CBOpen(0L))
        {
        mustwrite = FALSE;

        /* Notify clipboard we have data */

        ior->io_Data    = (STRPTR)satisfy;
        ior->io_ClipID  = 0L;
        ior->io_Command = CBD_POST;
        DoIO( (struct IORequest *) ior);

        postID = ior->io_ClipID;

        printf("\nClipID = %ld\n",postID);

        /* Wait for CTRL-C break, or message from clipboard */
        Wait(SIGBREAKF_CTRL_C|(1L << satisfy->mp_SigBit));

        /* see if we got a message, or a break */
        puts("Woke up");


        if (sm = (struct SatisfyMsg *)GetMsg(satisfy))
            {
            puts("Got a message from the clipboard\n");

            /* We got a message - we MUST write some data */
            mustwrite = TRUE;
            }
        else
            {
            /* Determine if we must write before exiting by
             * checking to see if our POST is still valid
             */

            ior->io_Command = CBD_CURRENTWRITEID;
            DoIO( (struct IORequest *) ior);

            printf("CURRENTWRITEID = %ld\n",ior->io_ClipID);

            if (postID >= ior->io_ClipID)
                mustwrite = TRUE;

            }

        /* Write the string of text */

        if (mustwrite)
            {
            if (!(CBWriteFTXT(ior,string)))
                puts("Error writing to clipboard");
            }
        else
            {
            puts("No need to write to clipboard");
            }

        CBClose(ior);
        }
    else
        {
        puts("Error opening clipboard.device");
        }

    DeletePort(satisfy);
    }
else
    {
    puts("Error creating message port");
    }

return(0);
}
```

```
/*
 * Changehook_Test.c
 *
 * Demonstrate the use of CBD_CHANGEHOOK command.
 * The program will set a hook and wait for the clipboard data to change.
 * You must put something in the clipboard in order for it to return.
 *
 * Compile with SAS C 5.10: LC -cfist -v -y -L+Hookface.o+cbio.o
 *
 * Requires Kickstart 36 or greater.
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/ports.h>
#include <exec/tasks.h>
#include <exec/io.h>
#include <devices/clipboard.h>
#include <dos/dos.h>
#include <utility/hooks.h>
#include "cb.h"

#include <clib/macros.h>
#include <clib/alib_protos.h>
#include <clib/exec_protos.h>

#include <stdio.h>
#include <string.h>


LONG version = 1L;

extern ULONG SysBase, DOSBase;

/* Data to pass around with the clipHook */
struct CHData
{
    struct Task *ch_Task;
    LONG ch_ClipID;
};

struct MsgPort *clip_port;
struct Hook hook;
struct CHData ch;

ULONG clipHook (struct Hook * h, VOID * o, struct ClipHookMsg * msg)
{
struct CHData *ch = (struct CHData *) h->h_Data;

if (ch)
   {
   /* Remember the ID of clip */
   ch->ch_ClipID = msg->chm_ClipID;

   /* Signal the task that started the hook */
   Signal (ch->ch_Task, SIGBREAKF_CTRL_E);
   }

return (0);
}

struct IOClipReq *OpenCB (LONG unit)
{
struct IOClipReq *clipIO;

/* Open clipboard unit 0 */

if (clipIO = CBOpen( 0L ))
    {
    ULONG hookEntry ();

    /* Fill out the IORequest */
    clipIO->io_Data = (char *) &hook;
    clipIO->io_Length = 1;
    clipIO->io_Command = CBD_CHANGEHOOK;

    /* Set up the hook data */
    ch.ch_Task = FindTask (NULL);

    /* Prepare the hook */
    hook.h_Entry = hookEntry;
    hook.h_SubEntry = clipHook;
    hook.h_Data = &ch;

    /* Start the hook */
    if (DoIO (clipIO))
        printf ("unable to set hook\n");
    else
        printf ("hook set\n");

    /* Return success */
    return ( clipIO );
    }

/* return failure */
return (NULL);
}

void CloseCB (struct IOClipReq *clipIO)
{

/* Fill out the IO request */
clipIO->io_Data = (char *) &hook;
clipIO->io_Length = 0;
clipIO->io_Command = CBD_CHANGEHOOK;

    /* Stop the hook */
if (DoIO (clipIO))
    printf ("unable to stop hook\n");
else
    /* Indicate success */
    printf ("hook is stopped\n");

CBClose(clipIO);
}

main (int argc, char **argv)
{
struct IOClipReq *clipIO;

ULONG sig_rcvd;

printf ("Test v%ld\n", version);

if (clipIO=OpenCB (0L))
    {
    sig_rcvd = Wait ((SIGBREAKF_CTRL_C | SIGBREAKF_CTRL_E));

    if (sig_rcvd & SIGBREAKF_CTRL_C)
        printf ("^C received\n");

    if (sig_rcvd & SIGBREAKF_CTRL_E)
        printf ("clipboard change, current ID is %ld\n", ch.ch_ClipID);

    CloseCB(clipIO);
    }
}
```

Support Functions Called from Example Programs

```
/* Cbio.c
 *
 * Provide standard clipboard device interface routines
 *            such as Open, Close, Post, Read, Write, etc.
 *
 * Compile with SAS C 5.10: LC -b1 -cfistq -v -y
 *
 *  NOTE - These functions are useful for writing, and reading simple
 *         FTXT.  Writing, and reading complex FTXT, ILBM, etc.,
 *         requires more work - under 2.0 it is highly recommended that
 *         you use iffparse.library.
 */

#include <exec/types.h>
#include <exec/ports.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <devices/clipboard.h>

#define CBIO 1

#include "cb.h"

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/****** cbio/CBOpen ************************************************
*
*   NAME
*       CBOpen() -- Open the clipboard.device
*
*   SYNOPSIS
*       ior = CBOpen(unit)
*
*       struct IOClipReq *CBOpen( ULONG )
*
*   FUNCTION
*       Opens the clipboard.device.  A clipboard unit number
*       must be passed in as an argument.  By default, the unit
*       number should be 0 (currently valid unit numbers are
*       0-255).
*
*   RESULTS
*       A pointer to an initialized IOClipReq structure, or
*       a NULL pointer if the function fails.
*
*********************************************************************/

struct IOClipReq *CBOpen(unit)
ULONG unit;
{
struct MsgPort *mp;
struct IOStdReq *ior;

if (mp = CreatePort(0L,0L))
    {
    if (ior=CreateExtIO(mp,sizeof(struct IOClipReq)))
        {
        if (!(OpenDevice("clipboard.device",unit,ior,0L)))
            {
            return((struct IOClipReq *)ior);
            }
        DeleteExtIO(ior);
        }
    DeletePort(mp);
    }
return(NULL);

}
```

```
/****** cbio/CBClose ***********************************************
*
*   NAME
*       CBClose() -- Close the clipboard.device
*
*   SYNOPSIS
*       CBClose()
*
*       void CBClose()
*
*   FUNCTION
*       Close the clipboard.device unit which was opened via
*       CBOpen().
*
*******************************************************************/

void CBClose(ior)
struct IOClipReq *ior;
{
struct MsgPort *mp;

mp = ior->io_Message.mn_ReplyPort;

CloseDevice((struct IOStdReq *)ior);
DeleteExtIO((struct IOStdReq *)ior);
DeletePort(mp);

}

/****** cbio/CBWriteFTXT *******************************************
*
*   NAME
*       CBWriteFTXT() -- Write a string of text to the clipboard.device
*
*   SYNOPSIS
*       success = CBWriteFTXT( ior, string)
*
*       int CBWriteFTXT(struct IOClipReq *, char *)
*
*   FUNCTION
*       Write a NULL terminated string of text to the clipboard.
*       The string will be written in simple FTXT format.
*
*       Note that this function pads odd length strings automatically
*       to conform to the IFF standard.
*
*   RESULTS
*       TRUE if the write succeeded, else FALSE.
*
*******************************************************************/

int CBWriteFTXT(ior,string)
struct IOClipReq *ior;
char *string;
{

ULONG length, slen;
BOOL odd;
int success;

slen = strlen(string);
odd = (slen & 1);               /* pad byte flag */

length = (odd) ? slen+1 : slen;

/* initial set-up for Offset, Error, and ClipID */

ior->io_Offset = 0;
ior->io_Error  = 0;
ior->io_ClipID = 0;


/* Create the IFF header information */

WriteLong(ior, (long *) "FORM");     /* "FORM"             */
length+=12L;                         /* + "[size]FTXTCHRS" */
WriteLong(ior, &length);               /* total length        */
WriteLong(ior, (long *) "FTXT");       /* "FTXT"              */
WriteLong(ior, (long *) "CHRS");       /* "CHRS"              */
WriteLong(ior, &slen);                 /* string length       */

/* Write string */
ior->io_Data    = (STRPTR)string;
ior->io_Length  = slen;
ior->io_Command = CMD_WRITE;
DoIO( (struct IORequest *) ior);

/* Pad if needed */
if (odd)
    {
    ior->io_Data   = (STRPTR)"";
    ior->io_Length = 1L;
    DoIO( (struct IORequest *) ior);
    }

/* Tell the clipboard we are done writing */

ior->io_Command=CMD_UPDATE;
DoIO( (struct IORequest *) ior);

/* Check if io_Error was set by any of the preceding IO requests */
success = ior->io_Error ? FALSE : TRUE;

return(success);
}

WriteLong(ior, ldata)
struct IOClipReq *ior;
long *ldata;
{

ior->io_Data    = (STRPTR)ldata;
ior->io_Length  = 4L;
ior->io_Command = CMD_WRITE;
DoIO( (struct IORequest *) ior);

if (ior->io_Actual == 4)
    {
    return( ior->io_Error ? FALSE : TRUE);
    }

return(FALSE);

}


/****** cbio/CBQueryFTXT *********************************************
*
*   NAME
*       CBQueryFTXT() -- Check to see if clipboard contains FTXT
*
*   SYNOPSIS
*       result = CBQueryFTXT( ior )
*
*       int CBQueryFTXT(struct IOClipReq *)
*
*   FUNCTION
*       Check to see if the clipboard contains FTXT.  If so,
*       call CBReadCHRS() one or more times until all CHRS
*       chunks have been read.
*
*   RESULTS
*       TRUE if the clipboard contains an FTXT chunk, else FALSE.
*
*   NOTES
*       If this function returns TRUE, you must either call
*       CBReadCHRS() until CBReadCHRS() returns FALSE, or
*       call CBReadDone() to tell the clipboard.device that
*       you are done reading.
*
*
*********************************************************************/
```

```
int CBQueryFTXT(ior)
struct IOClipReq *ior;
{
ULONG cbuff[4];


/* initial set-up for Offset, Error, and ClipID */

ior->io_Offset = 0;
ior->io_Error  = 0;
ior->io_ClipID = 0;

/* Look for "FORM[size]FTXT" */

ior->io_Command = CMD_READ;
ior->io_Data    = (STRPTR)cbuff;
ior->io_Length  = 12;

DoIO( (struct IORequest *) ior);


/* Check to see if we have at least 12 bytes */

if (ior->io_Actual == 12L)
    {
    /* Check to see if it starts with "FORM" */
    if (cbuff[0] == ID_FORM)
        {
        /* Check to see if its "FTXT" */
        if (cbuff[2] == ID_FTXT)
            return(TRUE);
        }

    /* It's not "FORM[size]FTXT", so tell clipboard we are done */
    }

CBReadDone(ior);

return(FALSE);
}


/****** cbio/CBReadCHRS ******************************************
*
*   NAME
*       CBReadCHRS() -- Reads the next CHRS chunk from clipboard
*
*   SYNOPSIS
*       cbbuf = CBReadCHRS( ior )
*
*       struct cbbuf *CBReadCHRS(struct IOClipReq * )
*
*   FUNCTION
*       Reads and returns the text in the next CHRS chunk
*       (if any) from the clipboard.
*
*       Allocates memory to hold data in next CHRS chunk.
*
*   RESULTS
*       Pointer to a cbbuf struct (see cb.h), or a NULL indicating
*       a failure (e.g., not enough memory, or no more CHRS chunks).
*
*       ***Important***
*
*       The caller must free the returned buffer when done with the
*       data by calling CBFreeBuf().
*
*   NOTES
*       This function strips NULL bytes, however, a full reader may
*       wish to perform more complete checking to verify that the
*       text conforms to the IFF standard (stripping data as required).
*
*       Under 2.0, the AllocVec() function could be used instead of
*       AllocMem() in which case the cbbuf structure may not be
*       needed.
*
*********************************************************************/
```

```
struct cbbuf *CBReadCHRS(ior)
struct IOClipReq *ior;
{
ULONG chunk,size;
struct cbbuf *buf;
int looking;

/* Find next CHRS chunk */

looking = TRUE;
buf = NULL;

while (looking)
      {
      looking = FALSE;

      if (ReadLong(ior,&chunk))
          {
          /* Is CHRS chunk ? */
          if (chunk == ID_CHRS)
              {
              /* Get size of chunk, and copy data */
              if (ReadLong(ior,&size))
                  {
                  if (size)
                      buf=FillCBData(ior,size);
                  }
              }

            /* If not, skip to next chunk */
          else
              {
              if (ReadLong(ior,&size))
                  {
                   looking = TRUE;
                   if (size & 1)
                       size++;    /* if odd size, add pad byte */

                    ior->io_Offset += size;
                  }
              }
          }
      }

if (buf == NULL)
    CBReadDone(ior);        /* tell clipboard we are done */

return(buf);
}


ReadLong(ior, ldata)
struct IOClipReq *ior;
ULONG *ldata;
{
ior->io_Command = CMD_READ;
ior->io_Data    = (STRPTR)ldata;
ior->io_Length  = 4L;

DoIO( (struct IORequest *) ior);

if (ior->io_Actual == 4)
    {
    return( ior->io_Error ? FALSE : TRUE);
    }

return(FALSE);
}
```

```
struct cbbuf *FillCBData(ior,size)
struct IOClipReq *ior;
ULONG size;
{
register UBYTE *to,*from;
register ULONG x,count;

ULONG length;
struct cbbuf *buf,*success;

success = NULL;

if (buf = AllocMem(sizeof(struct cbbuf),MEMF_PUBLIC))
    {

    length = size;
    if (size & 1)
        length++;            /* if odd size, read 1 more */

    if (buf->mem = AllocMem(length+1L,MEMF_PUBLIC))
        {
        buf->size = length+1L;

        ior->io_Command = CMD_READ;
        ior->io_Data    = (STRPTR)buf->mem;
        ior->io_Length  = length;

        to = buf->mem;
        count = 0L;

        if (!(DoIO( (struct IOStdReq *) ior)))
            {
            if (ior->io_Actual == length)
                {
                success = buf;      /* everything succeeded */

                /* strip NULL bytes */
                for (x=0, from=buf->mem ;x<size;x++)
                     {
                     if (*from)
                         {
                         *to = *from;
                         to++;
                         count++;
                         }

                     from++;
                     }
                *to=0x0;            /* Null terminate buffer */
                buf->count = count; /* cache count of chars in buf */
                }
            }

        if (!(success))
            FreeMem(buf->mem,buf->size);
        }
    if (!(success))
        FreeMem(buf,sizeof(struct cbbuf));
    }

return(success);
}
```

```
/****** cbio/CBReadDone *******************************************
*
*   NAME
*       CBReadDone() -- Tell clipboard we are done reading
*
*   SYNOPSIS
*       CBReadDone( ior )
*
*       void CBReadDone(struct IOClipReq * )
*
*   FUNCTION
*       Reads past end of clipboard file until io_Actual is equal to 0.
*       This is tells the clipboard that we are done reading.
*
*******************************************************************/

void CBReadDone(ior)
struct IOClipReq *ior;
{
char buffer[256];

ior->io_Command = CMD_READ;
ior->io_Data    = (STRPTR)buffer;
ior->io_Length  = 254;


/* falls through immediately if io_Actual == 0 */

while (ior->io_Actual)
      {
      if (DoIO( (struct IORequest *) ior))
          break;
      }
}

/****** cbio/CBFreeBuf *******************************************
*
*   NAME
*       CBFreeBuf() -- Free buffer allocated by CBReadCHRS()
*
*   SYNOPSIS
*       CBFreeBuf( buf )
*
*       void CBFreeBuf( struct cbbuf * )
*
*   FUNCTION
*       Frees a buffer allocated by CBReadCHRS().
*
*******************************************************************/

void CBFreeBuf(buf)
struct cbbuf *buf;
{
FreeMem(buf->mem, buf->size);
FreeMem(buf, sizeof(struct cbbuf));
}
```

```
****************************************************************************
*        Hookface.asm
*        assembly routines for Chtest
*
*        Assemble with Adapt  hx68 hookface.a to hookface.o
*        Link with Changehook_Test.o as shown in Changehook_Test.c header
*
****************************************************************************
        INCDIR  'include:'
        INCLUDE 'exec/types.i'
        INCLUDE 'utility/hooks.i'
        xdef    _callHook
        xdef    _callHookPkt
        xdef    _hookEntry
        xdef    _stubReturn
****************************************************************************
* new hook standard
* use struct Hook (with minnode at the top)
*
* *** register calling convention: ***
*       A0 - pointer to hook itself
*       A1 - pointer to parameter packed ("message")
*       A2 - Hook specific address data ("object," e.g, gadget )
*
* ***  C conventions: ***
* Note that parameters are in unusual register order: a0, a2, a1.
* This is to provide a performance boost for assembly language
* programming (the object in a2 is most frequently untouched).
* It is also no problem in "register direct" C function parameters.
*
* calling through a hook
*       callHook( hook, object, msgid, p1, p2, ... );
*       callHookPkt( hook, object, msgpkt );
*
* using a C function:   CFunction( hook, object, message );
*       hook.h_Entry = hookEntry;
*       hook.h_SubEntry = CFunction;
****************************************************************************
* C calling hook interface for prepared message packet
_callHookPkt:
        movem.l a2/a6,-(sp)     ; protect
        move.l  12(sp),a0       ; hook
        move.l  16(sp),a2       ; object
        move.l  20(sp),a1       ; message
        ; ------ now have registers ready, invoke function
        pea.l   hreturn(pc)
        move.l  h_Entry(a0),-(sp)       ; old rts-jump trick
        rts
hreturn:
        movem.l (sp)+,a2/a6
        rts

* C calling hook interface for "varargs message packet"
_callHook:
        movem.l a2/a6,-(sp)     ; protect
        move.l  12(sp),a0       ; hook
        move.l  16(sp),a2       ; object
        lea.l   20(sp),a1       ; message
        ; ------ now have registers ready, invoke function
        pea.l   hpreturn(pc)
        move.l  h_Entry(a0),-(sp)       ; old rts-jump trick
        rts
hpreturn:
        movem.l (sp)+,a2/a6
        rts

* entry interface for C code (large-code, stack parameters)
_hookEntry:
        move.l  a1,-(sp)
        move.l  a2,-(sp)
        move.l  a0,-(sp)
        move.l  h_SubEntry(a0),a0       ; C entry point
        jsr     (a0)
        lea     12(sp),sp
_stubReturn:
        rts
```

Include File for the Example Programs

```
/***********************************************************************
 *
 * cb.h -- Include file used by clipdemo.c, changehook_test.c and cbio.c
 *
 ***********************************************************************/

struct cbbuf {

        ULONG size;     /* size of memory allocation            */
        ULONG count;    /* number of characters after stripping */
        UBYTE *mem;     /* pointer to memory containing data    */
};

#define MAKE_ID(a,b,c,d) ((a<<24L) | (b<<16L) | (c<<8L) | d)

#define ID_FORM MAKE_ID('F','O','R','M')
#define ID_FTXT MAKE_ID('F','T','X','T')
#define ID_CHRS MAKE_ID('C','H','R','S')

#ifdef CBIO

/* prototypes */

struct IOClipReq        *CBOpen         ( ULONG );
void                    CBClose         (struct IOClipReq *);
int                     CBWriteFTXT     (struct IOClipReq *, char *);
int                     CBQueryFTXT     (struct IOClipReq *);
struct cbbuf            *CBReadCHRS     (struct IOClipReq *);
void                    CBReadDone      (struct IOClipReq *);
void                    CBFreeBuf       (struct cbbuf *);


/* routines which are meant to be used internally by routines in cbio */

int                     WriteLong       (struct IOClipReq *, long *);
int                     ReadLong        (struct IOClipReq *, ULONG *);
struct cbbuf            *FillCBData     (struct IOClipReq *, ULONG);

#else

/* prototypes */

extern struct IOClipReq *CBOpen         ( ULONG );
extern void             CBClose         (struct IOClipReq *);
extern int              CBWriteFTXT     (struct IOClipReq *, char *);
extern int              CBQueryFTXT     (struct IOClipReq *);
extern struct cbbuf     *CBReadCHRS     (struct IOClipReq *);
extern void             CBReadDone      (struct IOClipReq *);
extern void             CBFreeBuf       (struct cbbuf *);

#endif
```

Additional Information on the Clipboard Device

Additional programming information on the clipboard device can be found in the include files for the clipboard device, iffparse library and utility library, and the Autodocs for all three. They are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

Clipboard Device Information	
INCLUDES	devices/clipboard.h
	devices/clipboard.i
	libraries/iffparse.h
	libraries/iffparse.h
	utility/hooks.h
	utility/hooks.i
AUTODOCS	clipboard.doc
	iffparse.doc
	utility.doc

chapter four
CONSOLE DEVICE

The console device provides the text-oriented interface for Intuition windows. It acts like an enhanced ASCII terminal obeying many of the standard ANSI sequences as well as special sequences unique to the Amiga. The console device also provides a copy-and-paste facility and an internal character map to redraw a window when it is resized.

New Console Features for Version 2.0

Feature	Description
CONU_LIBRARY	New #define
CONU_STANDARD	New #define
CONU_CHARMAP	Console Unit
CONU_SNIPMAP	Console Unit
CONFLAG_DEFAULT	Console Flag
CONFLAG_NODRAW_ON_NEWSIZE	Console Flag

Compatibility Warning: The new features for the 2.0 console device are not backwards compatible.

Console Device Commands and Functions

Command	Operation
CD_ASKDEFAULTKEYMAP	Get the current default keymap.
CD_ASKKEYMAP	Get the current key map structure for this console.
CD_SETDEFAULTKEYMAP	Set the current default keymap.
CD_SETKEYMAP	Set the current key map structure for this console.
CMD_CLEAR	Remove any reports waiting to satisfy read requests from the console input buffer.
CMD_READ	Read the next input, generally from the keyboard. The form of this input is as an ANSI byte stream.
CMD_WRITE	Write a text record to the display interpreting any ANSI control characters in the record.

Console Device Function

CDInputHandler()	Handle an input event for the console device.
RawKeyConvert()	Decode raw input classes and convert input events of type IECLASS_RAWKEY to ANSI bytes based on the keymap in use.

Exec Functions as Used in This Chapter

AbortIO()	Abort an I/O request to the console device.
CheckIO()	Return the status of an I/O request.
CloseDevice()	Relinquish use of the console device. All requests must be complete before closing.
DoIO()	Initiate a command and wait for completion (synchronous request).
GetMsg()	Get the next message from the reply port.
OpenDevice()	Obtain use of the console device. You specify the type of unit and its characteristics in the call to **OpenDevice()**.
OpenLibrary()	Gain access to a library.
OpenWindow()	Open an intuition window.
SendIO()	Initiate a command and return immediately (asynchronous request).
Wait()	Wait for one or more signals.
WaitIO()	Wait for completion of an I/O request and remove it from the reply port.
WaitPort()	Wait for the reply port to be non-empty. Does not remove the message from port.

Exec Support Functions as Used in This Chapter

CreateExtIO	Create an extended I/O request structure for use in communicating with the console device.
CreatePort()	Create a message port for reply messages from the console device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete the extended I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The console device operates like the other Amiga devices. To use it, you must first open the console device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The I/O request used by the console device is called **IOStdReq**.

```
struct IOStdReq
{
    struct  Message io_Message;
    struct  Device  *io_Device;        /* device node pointer  */
    struct  Unit    *io_Unit;          /* unit (driver private)*/
    UWORD   io_Command;                /* device command */
    UBYTE   io_Flags;
    BYTE    io_Error;                  /* error or warning num */
    ULONG   io_Actual;                 /* actual number of bytes transferred */
    ULONG   io_Length;                 /* requested number bytes transferred*/
    APTR    io_Data;                   /* points to data area */
    ULONG   io_Offset;                 /* offset for block structured devices */
};
```

See the include file *exec/io.h* for the complete structure definition.

CONSOLE DEVICE UNITS

The console device provides four units, three that require a console window and one that does not. The unit type is specified when you open the device. See the "Opening the Console Device" section below for more details.

The CONU_STANDARD unit (0) is generally used with a SMART_REFRESH window. This unit has the least amount of overhead (e.g., memory usage and rendering time), and is highly compatible with all versions of the operating system.

As of V36, a character mapped console device was introduced. There are two variations of character mapped console units. Both must be used with SIMPLE_REFRESH windows and both have the ability to automatically redraw a console window when resized or revealed.

A character mapped console can be opened which allows the user to drag-select text with the mouse and COPY the highlighted area. The copied text can then be PASTEd into other console windows or other windows which support reading data from the clipboard device.

Character mapped console units have more overhead than standard consoles (e.g., rendering times and memory usage).

The CONU_LIBRARY unit (-1) does not require a console window. It is designed to be primarily used with the console device functions and also with the console device commands that do not require a console window.

The Amiga uses the ECMA-94 Latin1 International 8-bit character set. See Appendix A (page 397) for a table of character codes.

OPENING THE CONSOLE DEVICE

Four primary steps are required to open the console device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an I/O request structure of type **IOStdReq**. The **IOStdReq** structure is created by the **CreateExtIO()** function. **CreateExtIO** will initialize your I/O request to point to your reply port.
- Open an Intuition window and set a pointer to it in the **io_Data** field of the **IOStdReq** and the size of the window in the **io_Length** field. This is the window to which the console will be attached. The window must be SIMPLE_REFRESH for use with the CONU_CHARMAP and CONU_SNIPMAP units.
- Open the console device. Call **OpenDevice()** passing it the I/O request and the type of console unit set in the **unit** and **flags** fields. Console unit types and flag values are listed below.

Console device units:

- CONU_LIBRARY – Return the device library vector pointer used for calling console device functions. No console is opened.
- CONU_STANDARD – Open a standard console.
- CONU_CHARMAP – Open a console with a character map.
- CONU_SNIPMAP – Open a console with a character map and copy-and-paste support.

See the include file *devices/conunit.h* for the unit definitions and the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for an explanation of each unit.

No Changes Required CONU_STANDARD has a numeric value of zero to insure compatibility with pre-V36 applications. CONU_LIBRARY has a numeric value of negative one and is also compatible with pre-V36 applications.

Console device flags:

- CONFLAG_DEFAULT – The console device will redraw the window when it is resized.
- CONFLAG_NODRAW_ON_NEWSIZE – The console device will not redraw the window when it is resized

The character map units, CONU_CHARMAP and CONU_SNIPMAP, are the only units which use the **flags** parameter to set how the character map is used. CONU_STANDARD units ignore the **flags** parameter.

See the include file *devices/conunit.h* for the flag definitions and the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for an explanation of the flags.

```
struct MsgPort *ConsoleMP;    /* Message port pointer */
struct IOStdReq *ConsIO;      /* I/O structure pointer */
struct Window   *win = NULL;  /* Window pointer */

struct NewWindow nw =
{
    10, 10,                      /* starting position (left,top) */
    620,180,                     /* width, height */
    -1,-1,                       /* detailpen, blockpen */
    CLOSEWINDOW,                 /* flags for idcmp */
    WINDOWDEPTH|WINDOWSIZING|
    WINDOWDRAG|WINDOWCLOSE|
    SIMPLE_REFRESH|ACTIVATE,      /* window flags */
    NULL,                        /* no user gadgets */
    NULL,                        /* no user checkmark */
    "Console Test",              /* title */
    NULL,                        /* pointer to window screen */
    NULL,                        /* pointer to super bitmap */
    100,45,                      /* min width, height */
    640,200,                     /* max width, height */
    WBENCHSCREEN                 /* open on workbench screen */
};

    /* Create reply port console */
if (!(ConsoleMP = CreatePort("RKM.Console",0)))
    cleanexit("Can't create write port\n",RETURN_FAIL);

    /* Create message block for device I/O */
if (!(ConsIO = CreateExtIO(ConsoleMP,sizeof(struct IOStdReq))))
    cleanexit("Can't create IORequest\n",RETURN_FAIL);

    /* Open a window --- we assume intuition.library is already open */
if (!(win = OpenWindow(&nw)))
    cleanexit("Can't open window\n",RETURN_FAIL);

    /* Set window pointer and size in I/O request */
ConsIO->io_Data = (APTR) win;
ConsIO->io_Length = sizeof(struct Window);

    /* Open the console device */
if (error = OpenDevice("console.device",CONU_CHARMAP,ConsIO,CONFLAG_DEFAULT))
    cleanexit("Can't open console.device\n",RETURN_FAIL);
```

CLOSING THE CONSOLE DEVICE

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**.

All I/O requests must be complete before **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()**.

```
if (!(CheckIO(ConsIO)))
    AbortIO(ConsIO);          /* Ask device to abort any pending requests */

WaitIO(ConsIO);               /* Wait for abort, then clean up */
CloseDevice(ConsIO);          /* Close console device */
```

About Console I/O

The console device may be thought of as a kind of terminal. You send character streams to the console device; you also receive them from the console device. These streams may be characters, control sequences or a combination of the two.

Console I/O is closely associated with the Amiga Intuition interface; a console must be tied to a window that is already opened. From the **Window** data structure, the console device determines how many characters it can display on a line and how many lines of text it can display in a window without clipping at any edge.

You can open the console device many times, if you wish. The result of each open call is a new console unit. AmigaDOS and Intuition see to it that only one window is currently active and its console, if any, is the only one (with a few exceptions) that receives notification of input events, such as keystrokes. Later in this chapter you will see that other Intuition events can be sensed by the console device as well.

> *Introducing...* For this entire chapter the characters "<CSI>" represent the *control sequence introducer*. For output you may use either the two-character sequence <Esc>[(0x1B 0x5B) or the one-byte value 0x9B. For input you will receive 0x9B unless the sequence has been typed by the user.

EXEC FUNCTIONS AND THE CONSOLE DEVICE

The various Exec functions such as **DoIO()**, **SendIO()**, **AbortIO()** and **CheckIO()** operate normally. The only caveats are that CMD_WRITE may cause your application to wait internally, even with **SendIO()**, and a task using CMD_READ to wait on a response from a console is at the user's mercy. If the user never reselects that window, and the console response provides the only wake-up call, that task will sleep forever.

GENERAL CONSOLE SCREEN OUTPUT

Console character screen output (as compared to console command sequence transmission) outputs all standard printable characters (character values hex 20 through 7E and A0 through FF) normally.

Many control characters such as BACKSPACE (0x8) and RETURN (0x13) are translated into their exact ANSI equivalent actions. The LINEFEED character (0xA) is a bit different in that it can be translated into a RETURN/LINEFEED action. The net effect is that the cursor moves to the first column of the next line whenever an <LF> is displayed. This option is set via the mode control sequences discussed under "Control Sequences for Window Output."

CONSOLE KEYBOARD INPUT

If you read from the console device, the keyboard inputs are preprocessed for you and you will get ASCII characters, such as "B." Most normal text-gathering programs will read from the console device in this manner. Some programs may also ask to receive raw events in their console stream. Keypresses are converted to ASCII characters or CSI sequences via the keymap associated with the unit.

Writing to the Console Device

You write to the console device by passing an I/O request to the device with a pointer to the write buffer set in **io_Data**, the number of bytes in the buffer set in **io_Length** and CMD_WRITE set in **io_Command.**

```
UBYTE *outstring= "Make it so.";

ConsIO->io_Data = outstring;
ConsIO->io_Length = strlen(outstring);
ConsIO->io_Command = CMD_WRITE;
DoIO(ConsIO);
```

You may also send NULL-terminated strings to the console device in the same manner except that **io_Length** must be set to -1.

```
ConsIO->io_Data = "\033[3mOh boy.";
ConsIO->io_Length = -1;
ConsIO->io_Command = CMD_WRITE;
DoIO(ConsIO);
```

The fragment above will output the string "Oh boy." in italics. Keep in mind that setting the text rendition to italics will remain in effect until you specifically instruct the console device to change it to another text style.

HINTS FOR WRITING TEXT

256 Is A Nice Round Number

You must keep in mind that the console device locks all layers while writing text. To avoid, problems with this, it is best to send smaller rather larger numbers of character to be written. We recommend no more than 256 bytes per write as the optimum size

Turn Off The Cursor

If your console is attached to a V1.2/1.3 SuperBitmap window, you will not see a cursor rendered. For output speed and compatibility with future OS versions which may visibly render the cursor, you should send the cursor-off sequence (ESC[0 p) whenever you open or reset (ESCc) a SuperBitmap window's console.

CONTROL SEQUENCES FOR WINDOW OUTPUT

The following table lists functions that the console device supports, along with the character stream that you must send to the console to produce the effect. For more information on the control sequences, consult the console.doc of the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.* The table uses the second form of <CSI>, that is, the hex value 0x9B, to minimize the number of characters to be transmitted to produce a function.

A couple of notes about the table. If an item is enclosed in square brackets, it is optional and may be omitted. For example, for INSERT [N] CHARACTERS the value for N is shown as optional. The console device responds to such optional items by treating the value of N as 1 if it is not specified. The value of N or M is always a decimal number, having one or more ASCII digits to express its value.

ANSI Console Control Sequences

Console Command	Sequence of Characters (in Hexadecimal Form)
BELL (Flash the display—do an Intuition **DisplayBeep()**)	07
BACKSPACE (move left one column)	08
HORIZONTAL TAB (move right one tab stop)	09
LINEFEED (move down one text line as specified by the mode function)	0A
VERTICAL TAB (move up one text line)	0B
FORMFEED (clear the console's window)	0C
CARRIAGE RETURN (move to first column)	0D
SHIFT IN (undo SHIFT OUT)	0E
SHIFT OUT (set MSB of each character before displaying)	0F
ESC (escape; can be part of the control sequence introducer)	1B
INDEX (move the active position down one line)	84
NEXT LINE (go to the beginning of the next line)	85
HORIZONTAL TABULATION SET (Set a tab at the active cursor position)	88
REVERSE INDEX (move the active position up one line)	8D
CSI (control sequence introducer)	9B
RESET TO INITIAL STATE	1B 63
INSERT [N] CHARACTERS (insert one or more spaces, shifting the remainder of the line to the right)	9B [N] 40
CURSOR UP [N] CHARACTER POSITIONS (default = 1)	9B [N] 41
CURSOR DOWN [N] CHARACTER POSITIONS (default = 1)	9B [N] 42
CURSOR FORWARD [N] CHARACTER POSITIONS (default = 1)	9B [N] 43
CURSOR BACKWARD [N] CHARACTER (default = 1)	9B [N] 44

CURSOR NEXT LINE [N] (to column 1)	9B [N] 45
CURSOR PRECEDING LINE [N] (to column 1)	9B [N] 46
CURSOR POSITION (where N is row, M is column, and semicolon (hex 3B) must be present as a separator, or if row is left out, so the console device can tell that the number after the semicolon actually represents the column number)	9B [N] [3B M] 48
CURSOR HORIZONTAL TABULATION (move cursor forward to Nth tab position)	9B [N] 49
ERASE IN DISPLAY (only to end of display)	9B 4A
ERASE IN LINE (only to end of line)	9B 4B
INSERT LINE (above the line containing the cursor)	9B 4C
DELETE LINE (remove current line, move all lines up one position to fill gap, blank bottom line)	9B 4D
DELETE CHARACTER [N] (that cursor is sitting on and to the right if [N] is specified)	9B [N] 50
SCROLL UP [N] LINES (Remove line(s) from top of window, move all other lines up, blanks [N] bottom lines)	9B [N] 53
SCROLL DOWN [N] LINES (Remove line(s) from bottom of window, move all other lines down, blanks [N] top lines)	9B [N] 54
CURSOR TABULATION CONTROL (where N = 0 set tab, 2 = clear tab, 5 = clear all tabs.)	9B [N] 57
CURSOR BACKWARD TABULATION (move cursor backward to Nth tab position.)	9B [N] 5A
SET LINEFEED MODE (cause LINEFEED to respond as RETURN-LINEFEED)	9B 32 30 68
RESET NEWLINE MODE (cause LINEFEED to respond only as LINEFEED)	9B 32 30 6C
DEVICE STATUS REPORT (cause console device to insert a CURSOR POSITION REPORT into your read stream ; see "Reading from the Console Device" for more information)	9B 36 6E
SELECT GRAPHIC RENDITION (select text style, character color, character cell color, background color)	9B N 3B 3N 3B 4N 3B >N 6D See note below.

For SELECT GRAPHIC RENDITION, any number of parameters, in any order, are valid. They are separated by semicolons.

The parameters follow:

<text style> =

0	Plain text	8	Concealed mode
1	Boldface	22	Normal color, not bold (V36)
2	faint (secondary color)	23	Italic off (V36)
3	Italic	24	Underscore off (V36)
4	Underscore	27	Reversed off (V36)
7	Reversed character/cell colors	28	Concealed off (V36)

<character color> =

30–37	System colors 0–7 for character color.
39	Reset to default character color

Transmitted as two ASCII characters.

<character cell color> =

40–47	System colors 0–7 for character cell color.
39	Reset to default character color

Transmitted as two ASCII characters.

<background color> =

>0–7	System colors 0–7 for background color. (V36)

You must specify the ">" in order for this to be recognized and it must be the last parameter.

For example, to select bold face, with color 3 as the character color, and color 0 as the character cell color and the background color, send the hex sequence:

```
9B 31 3B 33 33 3B 34 30 3B 3E 30 6D
```

representing the ASCII sequence:

```
<CSI>1;33;40;>0m
```

where <CSI> is the control sequence introducer, here used as the single character value 0x9B.

Go Easy On The Eyes. In most cases, the character cell color and the background color should be the same.

Set Graphic Rendition Implementation Notes

Previous versions of the operating system did not support the global background color sequence as is listed above. Instead, the background color was set by setting the character cell color and then clearing the screen (e.g., a FORMFEED).

In fact, vacated areas of windows (vacated because of an ERASE or SCROLL) were filled in with the character cell color. This is no longer the case. Now, when an area is vacated, it is filled in with the global background color.

SMART_REFRESH windows are a special case:

Under V33–V34:

The cell color had to be set and a FORMFEED (clear window) needed to be sent on resize or immediately to clear the window and set the background color.

For example, if you took a CLI window and sent the sequence to set the cell color to something other than the default, the background color would not be changed immediately (contrary to what was expected).

If you then sent a FORMFEED, the background color would change, but if you resized the window larger, you would note that the newly revealed areas were filled in with PEN 0.

Under V36–V37 (non-character mapped):

You need to set the global background color and do a FormFeed. The background color will then be used to fill the window, but like V33-V34, if you make the window larger, the vacated areas will be filled in with PEN 0.

Under V36–V37 (character mapped):

You need to set the global background color, the window is redrawn immediately (because we have the character map) and will be correctly redrawn with the global background color on subsequent resizes.

The sequences in the next table are not ANSI standard sequences, they are private Amiga sequences. In these command descriptions, length, width, and offset are comprised of one or more ASCII digits, defining a decimal value.

Amiga Console Control Sequences

Console Command	**Sequence of Characters** (in Hexadecimal Form)
ENABLE SCROLL (this is the default)	9B 3E 31 68
DISABLE SCROLL	9B 3E 31 6C
AUTOWRAP ON (the default)	9B 3F 37 68
AUTOWRAP OFF	9B 3F 37 6C
SET PAGE LENGTH (in character raster lines, causes console to recalculate, using current font, how many text lines will fit on the page)	9B <length> 74
SET LINE LENGTH (in character positions, using current font, how many characters should be placed on each line)	9B <width> 75
SET LEFT OFFSET (in raster columns, how far from the left of the window should the text begin)	9B <offset> 78

SET TOP OFFSET (in raster lines, how far from the top of the window's **RastPort** should the topmost line of the character begin)	9B <offset> 79
SET RAW EVENTS (set the raw input events that will trigger an INPUT EVENT REPORT. see the "Selecting Raw Input Events" section below for more details.)	9B <events> 7B
INPUT EVENT REPORT (returned by the console device in response to a raw event set by the SET RAW EVENT sequence. see the "Input Event Reports" section below for more details.)	9B <parameters> 7C
RESET RAW EVENTS (reset the raw events set by the SET RAW EVENT sequence. see the "Selecting Raw Input Events" section below.)	9B <events> 7D
SPECIAL KEY REPORT (returned by the console device whenever HELP, or one of the function keys or arrow keys is pressed. Some sequences do not end with 7E)	9B <keyvalue> 7E
SET CURSOR RENDITION (make the cursor visible or invisible: Note—turning off the cursor increases text output speed)	9B N 20 70
Invisible:	9B 30 20 70
Visible:	9B 20 70
WINDOW STATUS REQUEST (ask the console device to tell you the current bounds of the window, in upper and lower row and column character positions. User may have resized or repositioned it. See "Window Bounds Report" below.)	9B 30 20 71
WINDOW BOUNDS REPORT (returned by the console device in response to a WINDOW STATUS REQUEST sequence)	9B 31 3B 31 3B <bot margin> 3B <right margin> 72
RIGHT AMIGA V PRESS (returned by the console device when the user presses RIGHT-AMIGA-V. See the "Copy and Paste Support" section below for more details.)	9B 30 20 76

Give Back What You Take. The console device normally handles the SET PAGE LENGTH, SET LINE LENGTH, SET LEFT OFFSET, and SET TOP OFFSET functions automatically. To allow it to do so again after setting your own values, send the functions without a parameter.

EXAMPLE CONSOLE CONTROL SEQUENCES

Move cursor right by 1:

Character string equivalents:

```
<CSI>C or
<CSI>1C
```

Numeric (hex) equivalents:

```
9B 43
9B 31 43
```

Move cursor right by 20:

Character string equivalent:

```
<CSI>20C
```

Numeric (hex) equivalent:

```
9B 32 30 43
```

Move cursor to upper-left corner (home):

Character string equivalents:

```
<CSI>H  or
<CSI>1;1H or
<CSI>;1H or
<CSI>1;H
```

Numeric (hex) equivalents:

```
9B 48
9B 31 3B 31 48
9B 3B 31 48
9B 31 3B 48
```

Move cursor to the fourth column of the first line of the window:

Character string equivalents:

```
<CSI>1;4H or
<CSI>;4H
```

Numeric (hex) equivalents:

```
9B 31 3B 34 48
9B 3B 34 48
```

Clear the window:

Character string equivalents:

```
<FF> or CTRL-L (clear window) or
<CSI>H<CSI>J (home and clear to end of window)
```

Numeric (hex) equivalents:

```
0C
9B 48 9B 4A
```

Reading from the Console Device

Reading input from the console device returns an ANSI 3.64 standard byte stream. This stream may contain normal characters and/or RAW input event information. You may also request other RAW input events using the SET RAW EVENTS and RESET RAW EVENTS control sequences discussed below. See "Selection of Raw Input Events."

Generally, console reads are performed asynchronously so that your program can respond to other events and other user input (such as menu selections) when the user is not typing on the keyboard. To perform asynchronous I/O, an I/O request is sent to the console using the **SendIO()** function (rather than a synchronous **DoIO()** which would wait until the read request returned with a character).

You read from the console device by passing an I/O request to the device with a pointer to the read buffer set in **io_Data**, the number of bytes in the buffer set in **io_Length** and CMD_READ set in **io_Command.**

```
#define READ_BUFFER_SIZE 25
char ConsoleReadBuffer[READ_BUFFER_SIZE];

ConsIO->io_Data = (APTR)ConsoleReadBuffer;
ConsIO->io_Length = READ_BUFFER_SIZE;
ConsIO->io_Command = CMD_READ;
SendIO(ConsIO);
```

> *You May Get Less Than You Bargained For.* A request for more than one character may be satisfied by the receipt of only one character. If you request more than one character, you will have to examine the **io_Actual** field of the request when it returns to determine how many characters you have actually received.

After sending the read request, your program can wait on a combination of signal bits including that of the reply port you created. The following fragment demonstrates waiting on both a queued console read request, and Window IDCMP messages:

```
ULONG conreadsig = 1 << ConsoleMP->mp_SigBit;
ULONG windowsig = 1 << win->UserPort->mp_SigBit;

/* A character, or an IDCMP msg, or both will wake us up */
ULONG signals = Wait(conreadsig | windowsig);

if (signals & conreadsig)
    {
    /* Then check for a character */
    };

if (signals & windowsig)
    {
    /* Then check window messages */
    };
```

INFORMATION ABOUT THE INPUT STREAM

For the most part, keys whose keycaps are labeled with ANSI-standard characters will ordinarily be translated into their ASCII-equivalent character by the console device through the use of its keymap. Keymap information can be found in the "Keymap Library" chapter of the *Amiga ROM Kernel Reference Manual: Libraries.*

For keys other than those with normal ASCII equivalents, an escape sequence is generated and inserted into your input stream. For example, in the default state (no raw input events selected) the function, arrow and special keys (reserved for 101 key keyboards) will cause the sequences shown in the next table to be inserted in the input stream.

Special Key Report Sequences

Key	Unshifted Sends	Shifted Sends	
F1	<CSI>0~	<CSI>10~	
F2	<CSI>1~	<CSI>11~	
F3	<CSI>2~	<CSI>12~	
F4	<CSI>3~	<CSI>13~	
F5	<CSI>4~	<CSI>14~	
F6	<CSI>5~	<CSI>15~	
F7	<CSI>6~	<CSI>16~	
F8	<CSI>7~	<CSI>17~	
F9	<CSI>8~	<CSI>18~	
F10	<CSI>9~	<CSI>19~	
F11	<CSI>20~	<CSI>30~	(101 key keyboard)
F12	<CSI>21~	<CSI>31~	(101 key keyboard)
HELP	<CSI>?~	<CSI>?~	(same sequence for both)
Insert	<CSI>40~	<CSI>50~	(101 key keyboard)
Page Up	<CSI>41~	<CSI>51~	(101 key keyboard)
Page Down	<CSI>42~	<CSI>52~	(101 key keyboard)
Pause/Break	<CSI>43~	<CSI>53~	(101 key keyboard)
Home	<CSI>44~	<CSI>54~	(101 key keyboard)
End	<CSI>45~	<CSI>55~	(101 key keyboard)
Arrow keys:			
Up	<CSI>A	<CSI>T	
Down	<CSI>B	<CSI>S	
Left	<CSI>D	<CSI> A	(notice the space
Right	<CSI>C	<CSI> @	after <CSI>)

CURSOR POSITION REPORT

If you have sent the DEVICE STATUS REPORT command sequence, the console device returns a cursor position report into your input stream. It takes the form:

```
<CSI><row>;<column>R
```

For example, if the cursor is at column 40 and row 12, here are the ASCII values (in hex) you receive in a stream:

```
9B 34 30 3B 31 32 52
```

WINDOW BOUNDS REPORT

A user may have either moved or resized the window to which your console is bound. By issuing a WINDOW STATUS REPORT to the console, you can read the current position and size in the input stream. This window bounds report takes the following form:

```
<CSI>1;1;<bottom margin>;<right margin> r
```

The bottom and right margins give you the window row and column dimensions as well. For a window that holds 20 lines with 60 characters per line, you will receive the following in the input stream:

```
9B 31 3B 31 3B 32 30 3B 36 30 20 72
```

Copy and Paste Support

As noted above, opening the console device with a unit of CONU_SNIPMAP allows the user to drag-select text with the mouse and copy the selection with Right-Amiga-C.

Internally, the snip is copied to a private buffer managed by the console device where it can be copied to other console device windows by pressing Right-Amiga-V.

However, your application should assume that the user is running the "Conclip" utility which is part of the standard Workbench 2.0 environment. Conclip copies snips from the console device to the clipboard device where they can be used by other applications which support reading from the clipboard.

When Conclip is running and the user presses Right-Amiga-V, the console device puts an escape sequence in your read stream—<CSI>0 v (Hex 9B 30 20 76)—which tells you that the user wants to paste text from the clipboard.

Upon receipt of this sequence, your application should read the contents of the clipboard device, make a copy of any text found there and then release the clipboard so that it can be used by other applications. See the "Clipboard Device" chapter for more information on reading data from it.

You paste what you read from the clipboard by using successive writes to the console. In order to avoid problems with excessively long data in the clipboard, you should limit the size of writes to something reasonable. (We define reasonable as no more than 1K per write with the ideal amount being 256 bytes.) You should also continue to monitor the console read stream for additional use input, paster requests and, possibly, RAW INPUT EVENTS while you are doing this.

You should *not* open a character mapped console unit with COPY capability if you are unable to support PASTE from the clipboard device. The user will reasonably expect to be able to PASTE into windows from which a COPY can be done.

Keep in mind that users do make mistakes, so an UNDO mechanism for aborting a PASTE is highly desirable—particularly if the user has just accidentally pasted text into an application like a terminal program which is sending data at a slow rate.

> *Use CON:, You'll Be Glad You Did.* It is highly recommended that you consider using the console-handler (CON:) if you want a console window with COPY and PASTE capablilities. CON: provides you with free PASTE support and is considerably easier to open and use than using the console device directly.

Selecting Raw Input Events

If the keyboard information–including "cooked" keystrokes–does not give you enough information about input events, you can request additional information from the console driver.

The command to SET RAW EVENTS is formatted as:

<CSI>[event-types-separated-by-semicolons]{

If, for example, you need to know when each key is pressed and released, you would request "RAW keyboard input." This is done by writing "<CSI>1{" to the console. In a single SET RAW EVENTS request, you can ask the console to set up for multiple event types at one time. You must send multiple numeric parameters, separating them by semicolons (;). For example, to ask for gadget pressed, gadget released, and close gadget events, write:

<CSI>7;8;11{

You can reset, that is, delete from reporting, one or more of the raw input event types by using the RESET RAW EVENTS command, in the same manner as the SET RAW EVENTS was used to establish them in the first place. This command stream is formatted as:

<CSI>[event-types-separated-by-semicolons]}

So, for example, you could reset all of the events set in the above example by transmitting the command sequence:

<CSI>7;8;11}

The Read Stream May Not Be Dry. There could still be pending RAW INPUT EVENTS in your read stream after turning off one or more RAW INPUT EVENTS.

The following table lists the valid raw input event types.

Raw Input Event Types

Request Number	Description	Number	Request Description
0	No-op (used internally)	11	Close Gadget
1	RAW keyboard input	12	Window resized
	Intuition swallows all	13	Window refreshed
	except the select button)	14	Preferences changed
2	RAW mouse input	15	Disk removed
3	Private Console Event	16	Disk inserted
4	Pointer position	17	Active window
5	(unused)	18	Inactive window
6	Timer	19	New pointer position (V36)
7	Gadget pressed	20	Menu help (V36)
8	Gadget released	21	Window changed (V36)
9	Requester activity		(zoom, move)
10	Menu numbers		

The event types—requester, window refreshed, active window, inactive window, window resized and window changed—are dispatched to the console unit which owns the window from which the events are generated, even if it is not the active (selected) window at the time the event is generated. This ensures that the proper console unit is notified of those events. All other events are dispatched to the active console unit (if it has requested those events).

Input Event Reports

If you select any of these events you will start to get information about the events in the following form:

<CSI><class>;<subclass>;<keycode>;<qualifiers>;<x>;<y>;<seconds>;<microseconds>|

<CSI>
is a one-byte field. It is the "control sequence introducer," 0x9B in hex.

<class>
is the RAW input event type, from the above table.

<subclass>
is usually 0. If the mouse is moved to the right controller, this would be 1.

<keycode>
indicates which raw key number was pressed. This field can also be used for mouse information.

The Raw Key Might Be The Wrong Key. National keyboards often have different keyboard arrangements. This means that a particular raw key number may represent different characters on different national keyboards. The normal console read stream (as opposed to raw events) will contain the proper ASCII character for the keypress as translated according to the user's keymap.

<qualifiers>
indicates the state of the keyboard and system.

The qualifiers are defined as follows:

Input Event Qualifiers

Bit	Mask	Key	
0	0001	Left shift	
1	0002	Right shift	
2	0004	Caps Lock	Associated keycode is special; see below.
3	0008	Ctrl	
4	0010	Left Alt	
5	0020	Right Alt	
6	0040	Left Amiga key pressed	
7	0080	Right Amiga key pressed	
8	0100	Numeric pad	
9	0200	Repeat	
10	0400	Interrupt	Not currently used.
11	0800	Multibroadcast	This window (active one) or all windows.
12	1000	Middle mouse button	(Not available on standard mouse)
13	2000	Right mouse button	
14	4000	Left mouse button	
15	8000	Relative mouse	Mouse coordinates are relative, not absolute.

The Caps Lock key is handled in a special manner. It generates a keycode only when it is pressed, not when it is released. However, the up/down bit (80 hex) is still used and reported. If pressing the Caps Lock key causes the LED to light, keycode 62 (Caps Lock pressed) is sent. If pressing the Caps Lock key extinguishes the LED, keycode 190 (Caps Lock released) is sent. In effect, the keyboard reports this key as held down until it is struck again.

The <x> and <y> fields are filled by some classes with an Intuition address: x<<16+y.

The <seconds> and <microseconds> fields contain the system time stamp taken at the time the event occurred. These values are stored as longwords by the system.

With RAW keyboard input selected, keys will no longer return a simple one-character "A" to "Z" but will instead return raw keycode reports of the form:

    <CSI>1;0;<keycode>;<qualifiers>;<prev1>;<prev2>;<seconds>;<microseconds>|

For example, if the user pressed and released the A key with the left Shift and right Amiga keys also pressed, you might receive the following data:

    <CSI>1;0;32;32769;14593;5889;421939940;316673|

    <CSI>1;0;160;32769;0;0;421939991;816683|

The <keycode> field is an ASCII decimal value representing the key pressed or released. Adding 128 to the pressed key code will result in the released keycode.

The <prev1> and <prev2> fields are relevant for the interpretation of keys which are modifiable by dead-keys (see "Dead-Class Keys" section). The <prev1> field shows the previous key pressed. The lower byte shows the qualifier, the upper byte shows the key code. The <prev2> field shows the key pressed before the previous key. The lower byte shows the qualifier, the upper byte shows the key code.

Using the Console Device Without a Window

Most console device processing involves a window, but there are functions and special commands that may be used without a window. To use the console device without a window, you call **OpenDevice()** with the console unit CONU_LIBRARY.

The console device functions are **CDInputHandler()** and **RawKeyConvert()**; they may only be used with the CONU_LIBRARY console unit. The console device commands which do not require a window are CD_ASKDEFAULTKEYMAP and CD_SETDEFAULTKEYMAP; they be used with any console unit. The advantage of using the commands with the CONU_LIBRARY unit is the lack of overhead required for CONU_LIBRARY because it doesn't require a window.

To use the functions requires the following steps:

- Declare the console device base address variable **ConsoleDevice** in the global data area.
- Declare storage for an I/O request of type **IOStdReq**.
- Open the console device with CONU_LIBRARY set as the console unit.
- Set the console device base address variable to point to the device library vector which is returned in **io_Device**.
- Call the console device function(s).
- Close the console device when you are finished.

```
#include <devices/conunit.h>
struct ConsoleDevice *ConsoleDevice;    /* declare device base address */

struct IOStdReq ConsIO= {0};               /* I/O request */

main()

    /* Open the device with CONU_LIBRARY for function use */
if (0 == OpenDevice("console.device",CONU_LIBRARY,(struct IORequest *)&ConsIO,0))
    {
    /* Set the base address variable to the device library vector */
    ConsoleDevice = (struct ConsoleDevice *)ConsIO.io_Device;

                    .
                    .    (console device functions would be called here)
                    .

    CloseDevice(ConsIO);
    }
```

The code fragment shows only the steps outlined above, it is not complete in any sense of the word. For a complete example of using a console device function, see the *rawkey.c* code example in the "Intuition: Mouse and Keyboard" chapter of the *Amiga ROM Kernel Reference Manual: Libraries.* The example uses the **RawKeyConvert()** function.

To use the commands with the CONU_LIBRARY console unit, you follow the same steps that were outlined in the "Opening the Console Device" section of this chapter.

```
struct MsgPort *ConsoleMP;       /* pointer to our message port */
struct IOStdReq *ConsoleIO;      /* pointer to our I/O request */
struct KeyMap  *keymap;          /* pointer to keymap */

    /* Create the message port */
if (ConsoleMP=CreateMsgPort())
    {
        /* Create the I/O request */
    if (ConsoleIO = CreateIORequest(ConsoleMP,sizeof(struct IOStdReq)))
        {
            /* Open the Console device */
        if (OpenDevice("console.device",CONU_LIBRARY,(struct IORequest *)ConsoleIO,0L))

            /* Inform user that it could not be opened */
            printf("Error: console.device did not open\n");
        else
            {
               /* Allocate memory for the keymap */
            if (keymap = (struct KeyMap *)
                    AllocMem(sizeof(struct KeyMap),MEMF_PUBLIC | MEMF_CLEAR))
                {
                /* device opened, send CD_ASKKEYMAP command to it */
                ConsoleIO->io_Length = sizeof(struct KeyMap);
                ConsoleIO->io_Data = (APTR)keymap;      /* where to put it */
                ConsoleIO->io_Command = CD_ASKKEYMAP;
                DoIO((struct IORequest *)ConsoleIO))
                }

            CloseDevice(ConsIO);
            }
```

Again, as in the previous code fragment, this is not complete (that's why it's a fragment!) and you should only use it as a guide.

Where Is All the Keymap Information?

Unlike previous editions of this chapter, this one has a very small amount of keymap information. Keymap information is now contained, appropriately enough, in the "Keymap Library" chapter of the *Amiga ROM Kernel Reference Manual: Libraries.*

Console Device Caveats

- Only one console unit can be attached per window. Sharing a console window must be done at a level higher than the device.
- Do not mix graphics.library calls with console rendering in the same areas of a window. It is permissible to send console sequences to adjust the area in which console renders, and use graphics.library calls to render outside of the area console is using.

For example, do not render text with console sequences and scroll using the graphics.library **ScrollRaster()** function.

- The character map feature is private and cannot be accessed by the programmer. Implementation details and behaviors of the character map my change in the future.
- Do not use an IDCMP with character mapped consoles. All Intuition messages should be obtained via RAW INPUT EVENTS from the console device.

Console Device Example Code

The following is a console device demonstration program with supporting routines:

```
/*
 * Console.c
 *
 * Example of opening a window and using the console device
 * to send text and control sequences to it.  The example can be
 * easily modified to do additional control sequences.
 *
 * Compile with SAS C 5.10: LC -b1 -cfistq -v -y -L
 *
 * Run from CLI only.
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#include <devices/console.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif
```

```
/* Note - using two character <CSI> ESC[.  Hex 9B could be used instead */
#define RESETCON  "\033c"
#define CURSOFF   "\033[0 p"
#define CURSON    "\033[ p"
#define DELCHAR   "\033[P"

* SGR (set graphic rendition) */
#define COLOR02   "\033[32m"
#define COLOR03   "\033[33m"
#define ITALICS   "\033[3m"
#define BOLD      "\033[1m"
#define UNDERLINE "\033[4m"
#define NORMAL    "\033[0m"


/* our functions */
void cleanexit(UBYTE *,LONG);
void cleanup(void);
BYTE OpenConsole(struct IOStdReq *,struct IOStdReq *, struct Window *);
void CloseConsole(struct IOStdReq *);
void QueueRead(struct IOStdReq *, UBYTE *);
UBYTE ConGetChar(struct MsgPort *, UBYTE *);
LONG ConMayGetChar(struct MsgPort *, UBYTE *);
void ConPuts(struct IOStdReq *, UBYTE *);
void ConWrite(struct IOStdReq *, UBYTE *, LONG);
void ConPutChar(struct IOStdReq *, UBYTE);
void main(int argc, char **argv);
struct NewWindow nw =
    {
    10, 10,                          /* starting position (left,top) */
    620,180,                         /* width, height */
    -1,-1,                           /* detailpen, blockpen */
    CLOSEWINDOW,                     /* flags for idcmp */
    WINDOWDEPTH|WINDOWSIZING|
    WINDOWDRAG|WINDOWCLOSE|
    SMART_REFRESH|ACTIVATE,          /* window flags */
    NULL,                            /* no user gadgets */
    NULL,                            /* no user checkmark */
    "Console Test",                  /* title */
    NULL,                            /* pointer to window screen */
    NULL,                            /* pointer to super bitmap */
    100,45,                          /* min width, height */
    640,200,                         /* max width, height */
    WBENCHSCREEN                     /* open on workbench screen */
    };


/* Opens/allocations we'll need to clean up */
struct Library  *IntuitionBase = NULL;
struct Window   *win = NULL;
struct IOStdReq *writeReq = NULL;   /* IORequest block pointer */
struct MsgPort  *writePort = NULL;  /* replyport for writes      */
struct IOStdReq *readReq = NULL;    /* IORequest block pointer */
struct MsgPort  *readPort = NULL;   /* replyport for reads       */
BOOL OpenedConsole = FALSE;

BOOL FromWb;

void main(argc, argv)
int argc;
char **argv;
    {
    struct IntuiMessage *winmsg;
    ULONG signals, conreadsig, windowsig;
    LONG lch;
    SHORT InControl = 0;
    BOOL Done = FALSE;
    UBYTE ch, ibuf;
    UBYTE obuf[200];
    BYTE error;

    FromWb = (argc==0L) ? TRUE : FALSE;

    if(!(IntuitionBase=OpenLibrary("intuition.library",0)))
         cleanexit("Can't open intuition\n",RETURN_FAIL);

    /* Create reply port and io block for writing to console */
```

```
if(!(writePort = CreatePort("RKM.console.write",0)))
     cleanexit("Can't create write port\n",RETURN_FAIL);

if(!(writeReq = (struct IOStdReq *)
                CreateExtIO(writePort,(LONG)sizeof(struct IOStdReq))))
     cleanexit("Can't create write request\n",RETURN_FAIL);

/* Create reply port and io block for reading from console */
if(!(readPort = CreatePort("RKM.console.read",0)))
     cleanexit("Can't create read port\n",RETURN_FAIL);

if(!(readReq = (struct IOStdReq *)
               CreateExtIO(readPort,(LONG)sizeof(struct IOStdReq))))
     cleanexit("Can't create read request\n",RETURN_FAIL);

/* Open a window */
if(!(win = OpenWindow(&nw)))
     cleanexit("Can't open window\n",RETURN_FAIL);

/* Now, attach a console to the window */
if(error = OpenConsole(writeReq,readReq,win))
     cleanexit("Can't open console.device\n",RETURN_FAIL);
else OpenedConsole = TRUE;

/* Demonstrate some console escape sequences */
ConPuts(writeReq,"Here's some normal text\n");
sprintf(obuf,"%s%sHere's text in color 3 and italics\n",COLOR03,ITALICS);
ConPuts(writeReq,obuf);
ConPuts(writeReq,NORMAL);
Delay(50);      /* Delay for dramatic demo effect */
ConPuts(writeReq,"We will now delete this asterisk =*=");
Delay(50);
ConPuts(writeReq,"\b\b");  /* backspace twice */
Delay(50);
ConPuts(writeReq,DELCHAR); /* delete the character */
Delay(50);

QueueRead(readReq,&ibuf); /* send the first console read request */

ConPuts(writeReq,"\n\nNow reading console\n");
ConPuts(writeReq,"Type some keys.  Close window when done.\n\n");

conreadsig = 1 << readPort->mp_SigBit;
windowsig = 1 << win->UserPort->mp_SigBit;

while(!Done)
    {
    /* A character, or an IDCMP msg, or both could wake us up */
    signals = Wait(conreadsig|windowsig);

    /* If a console signal was received, get the character */
    if (signals & conreadsig)
        {
        if((lch = ConMayGetChar(readPort,&ibuf)) != -1)
            {
            ch = lch;
            /* Show hex and ascii (if printable) for char we got.
             * If you want to parse received control sequences, such as
             * function or Help keys, you would buffer control sequences
             * as you receive them, starting to buffer whenever you
             * receive 0x9B (or 0x1B[ for user-typed sequences) and
             * ending when you receive a valid terminating character
             * for the type of control sequence you are receiving.
             * For CSI sequences, valid terminating characters
             * are generally 0x40 through 0x7E.
             * In our example, InControl has the following values:
             * 0 = no, 1 = have 0x1B, 2 = have 0x9B OR 0x1B and [,
             * 3 = now inside control sequence, -1 = normal end esc,
             * -2 = non-CSI(no [) 0x1B end esc
             * NOTE - a more complex parser is required to recognize
             *  other types of control sequences.
             */

            /* 0x1B ESC not followed by '[', is not CSI seq */
            if (InControl==1)
                {
                if(ch=='[') InControl = 2;
```

```
                    else InControl = -2;
                    }

                if ((ch==0x9B)||(ch==0x1B))  /* Control seq starting */
                    {
                    InControl = (ch==0x1B) ? 1 : 2;
                    ConPuts(writeReq,"=== Control Seq ===\n");
                    }

                /* We'll show value of this char we received */
                if (((ch >= 0x1F)&&(ch <= 0x7E))||(ch >= 0xA0))
                   sprintf(obuf,"Received: hex %02x = %c\n",ch,ch);
                else sprintf(obuf,"Received: hex %02x\n",ch);
                ConPuts(writeReq,obuf);

                /* Valid ESC sequence terminator ends an ESC seq */
                if ((InControl==3)&&((ch >= 0x40) && (ch <= 0x7E)))
                    {
                    InControl = -1;
                    }
                if (InControl==2) InControl = 3;
                /* ESC sequence finished (-1 if OK, -2 if bogus) */
                if (InControl < 0)
                    {
                    InControl = 0;
                    ConPuts(writeReq,"=== End Control ===\n");
                    }
                }
            }

        /* If IDCMP messages received, handle them */
        if (signals & windowsig)
            {
            /* We have to ReplyMsg these when done with them */
            while (winmsg = (struct IntuiMessage *)GetMsg(win->UserPort))
                {
                switch(winmsg->Class)
                    {
                    case CLOSEWINDOW:
                      Done = TRUE;
                      break;
                    default:
                      break;
                     }
                ReplyMsg((struct Message *)winmsg);
                }
            }
        }

    /* We always have an outstanding queued read request
     * so we must abort it if it hasn't completed,
     * and we must remove it.
     */
    if(!(CheckIO(readReq)))  AbortIO(readReq);
    WaitIO(readReq);     /* clear it from our replyport */

    cleanup();
    exit(RETURN_OK);
    }

void cleanexit(UBYTE *s,LONG n)
    {
    if(*s & (!FromWb)) printf(s);
    cleanup();
    exit(n);
    }

void cleanup()
    {
    if(OpenedConsole) CloseConsole(writeReq);
    if(readReq)       DeleteExtIO(readReq);
    if(readPort)      DeletePort(readPort);
    if(writeReq)      DeleteExtIO(writeReq);
    if(writePort)     DeletePort(writePort);
    if(win)           CloseWindow(win);
    if(IntuitionBase) CloseLibrary(IntuitionBase);
    }
```

```
/* Attach console device to an open Intuition window.
 * This function returns a value of 0 if the console
 * device opened correctly and a nonzero value (the error
 * returned from OpenDevice) if there was an error.
 */
BYTE OpenConsole(writereq, readreq, window)
struct IOStdReq *writereq;
struct IOStdReq *readreq;
struct Window *window;
    {
    BYTE error;

    writereq->io_Data = (APTR) window;
    writereq->io_Length = sizeof(struct Window);
    error = OpenDevice("console.device", 0, writereq, 0);
    readreq->io_Device = writereq->io_Device; /* clone required parts */
    readreq->io_Unit   = writereq->io_Unit;
    return(error);
    }

void CloseConsole(struct IOStdReq *writereq)
    {
    CloseDevice(writereq);
    }

/* Output a single character to a specified console
 */
void ConPutChar(struct IOStdReq *writereq, UBYTE character)
    {
    writereq->io_Command = CMD_WRITE;
    writereq->io_Data = (APTR)&character;
    writereq->io_Length = 1;
    DoIO(writereq);
    /* command works because DoIO blocks until command is done
     * (otherwise ptr to the character could become invalid)
     */
    }


/* Output a stream of known length to a console
 */
void ConWrite(struct IOStdReq *writereq, UBYTE *string, LONG length)
    {
    writereq->io_Command = CMD_WRITE;
    writereq->io_Data = (APTR)string;
    writereq->io_Length = length;
    DoIO(writereq);
    /* command works because DoIO blocks until command is done
     * (otherwise ptr to string could become invalid in the meantime)
     */
    }


/* Output a NULL-terminated string of characters to a console
 */
void ConPuts(struct IOStdReq *writereq,UBYTE *string)
    {
    writereq->io_Command = CMD_WRITE;
    writereq->io_Data = (APTR)string;
    writereq->io_Length = -1;  /* means print till terminating null */
    DoIO(writereq);
    }

/* Queue up a read request to console, passing it pointer
 * to a buffer into which it can read the character
 */
void QueueRead(struct IOStdReq *readreq, UBYTE *whereto)
   {
   readreq->io_Command = CMD_READ;
   readreq->io_Data = (APTR)whereto;
   readreq->io_Length = 1;
   SendIO(readreq);
   }
```

```
/* Check if a character has been received.
 * If none, return -1
 */
LONG ConMayGetChar(struct MsgPort *msgport, UBYTE *whereto)
    {
    register temp;
    struct IOStdReq *readreq;

    if (!(readreq = (struct IOStdReq *)GetMsg(msgport))) return(-1);
    temp = *whereto;                /* get the character */
    QueueRead(readreq,whereto);     /* then re-use the request block */
    return(temp);
    }

/* Wait for a character
 */
UBYTE ConGetChar(struct MsgPort *msgport, UBYTE *whereto)
    {
    register temp;
    struct IOStdReq *readreq;

    WaitPort(msgport);
    readreq = (struct IOStdReq *)GetMsg(msgport);
    temp = *whereto;               /* get the character */
    QueueRead(readreq,whereto);    /* then re-use the request block*/
    return((UBYTE)temp);
    }
```

Additional Information on the Console Device

Additional programming information on the console device can be found in the include files and the Autodocs for the console device. Both are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Console Device Information	
INCLUDES	devices/console.h devices/console.i devices/conunit.h devices/conunit.i
AUTODOCS	console.doc

chapter five
GAMEPORT DEVICE

The gameport device manages access to the Amiga gameport connectors for the operating system. It enables the Amiga to interface with various external pointing devices like mice (two and three button), joysticks, trackballs and light pens. There are two units in the gameport device, unit 0 and unit 1.

Amiga Gameport Connectors		
	Unit 0	**Unit 1**
A3000	Front Connector	Back Connector
A2000	Left Connector	Right Connector
A1000	1	2
A500	1 JOYSTICK	2 JOYSTICK

Gameport Device Commands and Functions

Command	Operation
CMD_CLEAR	Clear the gameport input buffer.
GPD_ASKCTYPE	Return the type of gameport controller being used.
GPD_ASKTRIGGER	Return the conditions that have been preset for triggering.
GPD_READEVENT	Read one or more gameport events.
GPD_SETCTYPE	Set the type of the controller to be used.
GPD_SETTRIGGER	Preset the conditions that will trigger a gameport event.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the gameport device.
CheckIO()	Return the status of an I/O request.
CloseDevice()	Relinquish use of the gameport device. All requests must be complete before closing.
DoIO()	Initiate a command and wait for completion (synchronous request).
OpenDevice()	Obtain shared use of one unit of the gameport device. The unit number specified is placed in the I/O request structure for use by gameport commands.
SendIO()	Initiate a command and return immediately (asynchronous request).
WaitIO()	Wait for the completion of an asynchronous request. When the request is complete the message will be removed from reply port.

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure of type **IOStdReq**. This structure will be used to communicate commands to the gameport device.
CreatePort()	Create a signal message port for reply messages from the gameport device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete an I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Who Runs The Mouse? When the input device or Intution is operating, unit 0 is usually dedicated to gathering mouse events. The input device uses the gameport device to read the mouse events. (For applications that take over the machine without starting up the input device or Intuition, unit 0 can perform the same functions as unit 1.) See the "Input Device" chapter for more information on the input device.

Device Interface

The gameport device operates like the other Amiga devices. To use it, you must first open the gameport device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The I/O request used by the gameport device is called **IOStdReq**.

```
struct IOStdReq
{
    struct  Message io_Message;
    struct  Device  *io_Device;      /* device node pointer  */
    struct  Unit    *io_Unit;        /* unit (driver private)*/
    UWORD   io_Command;              /* device command */
    UBYTE   io_Flags;
    BYTE    io_Error;                /* error or warning num */
    ULONG   io_Actual;               /* actual number of bytes transferred */
    ULONG   io_Length;               /* requested number bytes transferred*/
    APTR    io_Data;                 /* points to data area */
    ULONG   io_Offset;               /* offset for block structured devices */
};
```

See the include file *exec/io.h* for the complete structure definition.

OPENING THE GAMEPORT DEVICE

Three primary steps are required to open the gameport device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an I/O request structure of type **IOStdReq**. The **IOStdReq** structure is created by the **CreateExtIO()** function. **CreateExtIO()** will initialize the I/O request with your reply port.
- Open the gameport device. Call **OpenDevice()**, passing the I/O request and and indicating the unit you wish to use.

```
struct MsgPort *GameMP;    /* Message port pointer */
struct IOStdReq *GameIO;   /* I/O request pointer */

  /* Create port for gameport device communications */
if (!(GameMP = CreatePort("RKM_game_port",0)))
    cleanexit(" Error: Can't create port\n",RETURN_FAIL);

  /* Create message block for device I/O */
if (!(GameIO = CreateExtIO(GameMP,sizeof(struct IOStdReq))))
    cleanexit(" Error: Can't create I/O request\n",RETURN_FAIL);

  /* Open the right/back (unit 1, number 2) gameport.device unit */
if (error=OpenDevice("gameport.device",1,GameIO,0))
    cleanexit(" Error: Can't open gameport.device\n",RETURN_FAIL);
```

The gameport commands are unit specific. The unit number specified in the call to **OpenDevice()** determines which unit is acted upon.

GAMEPORT DEVICE CONTROLLERS

The Amiga has five gameport device controller types.

Gameport Device Controllers	
Controller Type	**Description**
GPCT_MOUSE	Mouse controller
GPCT_ABSJOYSTICK	Absolute (digital) joystick
GPCT_RELJOYSTICK	Relative (digital) joystick
GPCT_ALLOCATED	Custom controller
GPCT_NOCONTROLLER	No controller

To use the gameport device, you must define the type of device connected to the gameport and define how the device is to respond. The gameport device can be set up to return the controller status immediately or only when certain conditions have been met.

When a gameport device unit reponds to a request for input, it creates an input event. The contents of the input event will vary based on the type of device and the trigger conditions you have declared.

- A mouse controller can report input events for one, two, or three buttons and for positive or negative (x,y) movements. A trackball controller or car-driving controller is generally of the same type and can be declared as a mouse controller.
- An absolute joystick reports one single event for each change of its current location. If, for example, the joystick is centered and the user pushes the stick forward and holds it in that position, only one single forward-switch event will be generated.
- A relative joystick, on the other hand, is comparable to an absolute joystick with "autorepeat" installed. As long as the user holds the stick in a position other than centered, the gameport device continues to generate position reports.
- There is currently no system software support for proportional joysticks or proportional controllers (e.g., paddles). If you write custom code to read proportional controllers or other controllers (e.g., light pen) make certain that you issue GPD_SETCTYPE (explained below) with controller type GPCT_ALLOCATED to insure that other applications know the connector is being used.

GPCT_NOCONTROLLER. The controller type GPCT_NOCONTROLLER is not a controller at all, but a flag to indicate that the unit is not being used at the present time.

CLOSING THE GAMEPORT DEVICE

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**.

All I/O requests must be complete before **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()** and remove them with **WaitIO()**.

```
if (!(CheckIO(GameIO)))
    {
    AbortIO(GameIO);  /* Ask device to abort request, if pending */
    }
WaitIO((GameIO);    /* Wait for abort, then clean up */
CloseDevice(GameIO);
```

Gameport Events

A gameport event is an **InputEvent** structure which describes the following:

- The class of the event - always set to **IECLASS_RAWMOUSE** for the gameport device.
- The subclass of the event - 0 for the left port; 1 for the right port.
- The code - which button and its state. (No report = 0xFF)
- The qualifier - only button and relative mouse bits are set.
- The position - either a data address or mouse position count.
- The time stamp - delta time since last report, returned as frame count in **tv_secs** field.
- The next event - pointer to next event.

```
struct InputEvent GameEV
{
    struct InputEvent *ie_NextEvent;    /* next event */
    UBYTE     ie_Class;                 /* input event class */
    UBYTE     ie_SubClass;              /* subclass of the class */
    UWORD     ie_Code;                  /* input event code */
    UWORD     ie_Qualifier;             /* event qualifiers in effect */
       union
       {
         struct
         {
          WORD    ie_x;                 /* x position for the event */
          WORD    ie_y;                 /* y position for the event */
         }  ie_xy;
         APTR ie_addr;
       } ie_position;
    struct timeval ie_TimeStamp;        /* delta time since last report
}
```

See the include file *devices/inputevent.h* for the complete structure definition and listing of input event fields. ‘endverbatim

READING GAMEPORT EVENTS

You read gameport events by passing an I/O request to the device with GPD_READEVENT set in **io_Command,** the address of the **InputEvent** structure to store events set in **io_Data** and the size of the structure set in **io_Length**.

```
struct InputEvent  GameEV;
struct IOStdRequest *GameIO;  /* Must be initialized prior to using */

void send_read_request()
{
GameIO->io_Command = GPD_READEVENT;  /* Read events */
GameIO->io_Length = sizeof (struct InputEvent);
GameIO->io_Data = (APTR)&GameEV;     /* put events in GameEV*/
SendIO(GameIO);                     /* Asynchronous */
}
```

SETTING GAMEPORT EVENT TRIGGER CONDITIONS

You set the conditions that can trigger a gameport event by passing an I/O request to the device with GPD_SETTRIGGER set in **io_Command** and the address of a **GamePortTrigger** structure set in **io_Data**.

The information needed for gameport trigger setting is placed into a **GamePortTrigger** data structure which is defined in the include file *devices/gameport.h*.

```
struct GamePortTrigger
{
    UWORD    gpt_Keys;       /* key transition triggers */
    UWORD    gpt_Timeout;    /* time trigger (vertical blank units) */
    UWORD    gpt_XDelta;     /* X distance trigger */
    UWORD    gpt_YDelta;     /* Y distance trigger */
}
```

A few points to keep in mind with the GPD_SETTRIGGER command are:

- Setting **GPTF_UPKEYS** enables the reporting of upward transitions. Setting **GPTF_DOWNKEYS** enables the reporting of downward transitions. These flags may both be specified.
- The field **gpt_Timeout** specifies the time interval (in vertical blank units) between reports in the absence of another trigger condition. In other words, an event is generated every **gpt_Timeout** ticks. Vertical blank units may differ from country to country (e.g 60 Hz NTSC, 50 Hz PAL.) To find out the exact frequency use this code fragment:

  ```
  #include <exec/execbase.h>
  extern struct ExecBase *SysBase;

  UBYTE get_frequency(void)
  {
  return((UBYTE)SysBase->VBlankFrequency);
  }
  ```

- The **gpt_XDelta** and **gpt_YDelta** fields specify the x and y distances which, if exceeded, trigger a report.

For a mouse controller, you can trigger on a certain minimum-sized move in either the x or y direction, on up or down transitions of the mouse buttons, on a timed basis, or any combination of these conditions.

For example, suppose you normally signal mouse events if the mouse moves at least 10 counts in either the x or y directions. If you are moving the cursor to keep up with mouse movements and the user moves the mouse less than 10 counts, after a period of time you will want to update the position of the cursor to exactly match the mouse position. Thus the timed report of current mouse counts would be preferred. The following structure would be used:

```
#define XMOVE 10
#define YMOVE 10

struct GamePortTrigger GameTR =
{
    GPTF_UPKEYS | GPTF_DOWNKEYS,    /* trigger on all key transitions */
    1800,                           /* and every 36(PAL) or 30(NTSC) seconds */
    XMOVE,                          /* for any 10 in an x or y direction */
    YMOVE
};
```

For a joystick controller, you can select timed reports as well as button-up and button-down report trigger conditions. For an absolute joystick specify a value of one (1) for the **GameTR_XDelta** and **GameTR_YDelta** fields or you will not get any direction events. You set the trigger conditions by using the following code or its equivalent:

```
struct IOStdReq *GameIO;

void set_trigger_conditions(struct GamePortTrigger *GameTR)
{
GameIO->io_Command = GPD_SETTRIGGER;    /* set trigger conditions */
GameIO->io_Data = (APTR)GameTR;         /* from GameTR */
GameIO->io_Length = sizeof(struct GamePortTrigger);
DoIO(GameIO);
}
```

Triggers and Reads. If a task sets trigger conditions and does not ask for the position reports the gameport device will queue them up anyway. If the trigger conditions occur again and the gameport device buffer is filled, the additional triggers will be ignored until the buffer is read by a device read request (GPD_READEVENT) or a system CMD_CLEAR command flushes the buffer.

DETERMINING THE TRIGGER CONDITIONS

You determine the conditions required for triggering gameport events by passing an I/O request to the device with GPD_ASKTRIGGER set in **io_Command**, the length of the **GamePortTrigger** structure set in **io_Length** and the address of the structure set in **io_Data**. The gameport device will respond with the event trigger conditions currently set.

```
struct IOStdReq *GameIO;  /* Must be initialized prior to using */

struct GamePortTrigger GameTR;

void get_trigger_conditions(struct GamePortTrigger *GameTR)
{
GameIO->io_Command = GPD_ASKTRIGGER;    /* get type of triggers */
GameIO->io_Data = (APTR)GameTR;         /* place data here */
GameIO->io_Length= sizeof(GameTR);
DoIO(GameIO);
}
```

Setting and Reading the Controller Type

DETERMINING THE CONTROLLER TYPE

You determine the type of controller being used by passing an I/O request to the device with GPD_ASKCTYPE set in **io_Command**, 1 set in **io_Length** and the number of the unit set in **io_Unit**. The gameport device will respond with the type of controller being used.

```
struct IOStdReq *GameIO;  /* Must be initialized prior to using */

BYTE GetControllerType()
{
BYTE controller_type = 0;

GameIO->io_Command = GPD_ASKCTYPE;         /* get type of controller */
GameIO->io_Data = (APTR)&controller_type;  /* place data here */
GameIO->io_Length = 1;
DoIO(GameIO);
return (controller_type);
}
```

The BYTE value returned corresponds to one of the five controller types noted above.

SETTING THE CONTROLLER TYPE

You set the type of gameport controller by passing an I/O request to the device with GPD_SETCTYPE set in **io_Command**, 1 set in **io_Length** and the address of the byte variable describing the controller type set in **io_Data**.

The gameport device is a shared device; many tasks may have it open at any given time. Hence, a high level protocol has been established to prevent multiple tasks from reading the same unit at the same time.

Three Step Protocol for Using the Gameport Device

Step 1:

Send GPD_ASKCTYPE to the device and check for a GPCT_NOCONTROLLER return. *Never* issue GPD_SETCTYPE without checking whether the desired gameport unit is in use.

Step 2:

If GPCT_NOCONTROLLER is returned, you have access to the gameport. Set the allocation flag to GPCT_MOUSE, GPCT_ABSJOYSTICK or GPCT_RELJOYSTICK if you use a system supported controller, or GPCT_ALLOCATED if you use a custom controller.

```
struct IOStdReq *GameIO;  /* Must be initialized prior to using */

BOOL set_controller_type(type)
BYTE type;
{

BOOL success = FALSE;
BYTE controller_type = 0;

Forbid();                              /*critical section start */
GameIO->io_Command = GPD_ASKCTYPE;     /* inquire current status */
GameIO->io_Length = 1;
GameIO->io_Flags = IOF_QUICK;
GameIO->io_Data = (APTR)&controller_type; /* put answer in here */
DoIO(GameIO);

/* No one is using this device unit, let's claim it */
if (controller_type == GPCT_NOCONTROLLER)
    {
    GameIO->io_Command = GPD_SETCTYPE;/* set controller type */
    GameIO->io_Length = 1;
    GameIO->io_Data = (APTR)&type;  /* set to input param */
    DoIO( GameIO);
    success = TRUE;
    UnitOpened = TRUE;
    }
Permit(); /* critical section end */

/* success can be TRUE or FALSE, see above */
return(success);
}
```

Step 3:

The program must set the controller type back to GPCT_NOCONTROLLER upon exiting your program:

```
struct IOStdReq *GameIO;  /* Must be initialized prior to using */

void free_gp_unit()
{
BYTE type = GPCT_NOCONTROLLER;
GameIO->io_Command = GPD_SETCTYPE;  /* set controller type */
GameIO->io_Length = 1;
GameIO->io_Data = (APTR)&type;      /* set to unused */
DoIO( GameIO);
}
```

This three step protocol allows applications to share the gameport device in a system compatible way.

A Word About The Functions. The functions shown above are designed to be included in any application using the gameport device. The first function, set_controller_type(), would be the first thing done after opening the gameport device. The second function, free_gp_unit(), would be the last thing done before closing the device.

Joystick Example Program

```
/*
 * Absolute_Joystick.c
 *
 * Gameport device absolute joystick example
 *
 * Compile with SAS 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <exec/exec.h>
#include <dos/dos.h>
#include <devices/gameport.h>
#include <devices/inputevent.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>
```

```
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }      /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }   /* really */
#endif

#define JOY_X_DELTA (1)
#define JOY_Y_DELTA (1)
#define TIMEOUT_SECONDS (10)

extern struct ExecBase *SysBase;

/*-----------------------------------------------------------------------
** Routine to print out some information for the user.
*/
VOID printInstructions(VOID)
{
printf("\n >>> gameport.device Absolute Joystick Demo <<<\n\n");

if (SysBase->VBlankFrequency==60)
    printf(" Running on NTSC system (60 Hz).\n");
else if (SysBase->VBlankFrequency==50)
    printf(" Running on PAL system (50 Hz).\n");

printf( " Attach joystick to rear connector (A3000) and (A1000).\n"
        " Attach joystick to right connector (A2000).\n"
        " Attach joystick to left connector (A500).\n"
        " Then move joystick and click its button(s).\n\n"
        " To exit program press and release fire button 3 consecutive times. \n"
        " The program also exits if no activity occurs for 1 minute.\n\n");
}


/*-----------------------------------------------------------------------
** print out information on the event received.
*/
BOOL check_move(struct InputEvent *game_event)
{
WORD xmove, ymove;
BOOL timeout=FALSE;

xmove = game_event->ie_X;
ymove = game_event->ie_Y;

if (xmove == 1)
    {
    if (ymove == 1) printf("RIGHT DOWN\n");
    else if (ymove == 0) printf("RIGHT\n");
    else if (ymove ==-1) printf("RIGHT UP\n");
    else printf("UNKNOWN Y\n");
    }
else if (xmove ==-1)
    {
    if (ymove == 1) printf("LEFT DOWN\n");
    else if (ymove == 0) printf("LEFT\n");
    else if (ymove ==-1) printf("LEFT UP\n");
    else printf("UNKNOWN Y\n");
    }
else if (xmove == 0)
    {
    if (ymove == 1) printf("DOWN\n");
    /* note that 0,0 can be a timeout, or a direction release. */
    else if (ymove == 0)
        {
        if (game_event->ie_TimeStamp.tv_secs >=
                        (UWORD)(SysBase->VBlankFrequency) * TIMEOUT_SECONDS)
            {
            printf("TIMEOUT\n");
            timeout=TRUE;
            }
        else printf("RELEASE\n");
        }
    else if (ymove ==-1) printf("UP\n");
    else printf("UNKNOWN Y\n");
    }
```

```
else
    {
    printf("UNKNOWN X ");
    if (ymove == 1) printf("unknown action\n");
    else if (ymove == 0) printf("unknown action\n");
    else if (ymove ==-1) printf("unknown action\n");
    else printf("UNKNOWN Y\n");
    }

return(timeout);

}

/*----------------------------------------------------------------------
** send a request to the gameport to read an event.
*/
VOID send_read_request( struct InputEvent *game_event,
                        struct IOStdReq *game_io_msg)
{
game_io_msg->io_Command = GPD_READEVENT;
game_io_msg->io_Flags   = 0;
game_io_msg->io_Data    = (APTR)game_event;
game_io_msg->io_Length  = sizeof(struct InputEvent);
SendIO(game_io_msg);  /* Asynchronous - message will return later */
}

/*----------------------------------------------------------------------
** simple loop to process gameport events.
*/
VOID processEvents( struct IOStdReq *game_io_msg,
                    struct MsgPort  *game_msg_port)
{
BOOL timeout;
SHORT timeouts;
SHORT button_count;
BOOL  not_finished;
struct InputEvent game_event;   /* where input event will be stored */

/* From now on, just read input events into the event buffer,
** one at a time.  READEVENT waits for the preset conditions.
*/
timeouts = 0;
button_count = 0;
not_finished = TRUE;

while ((timeouts < 6) && (not_finished))
    {
    /* Send the read request */
    send_read_request(&game_event,game_io_msg);

    /* Wait for joystick action */
    Wait(1L << game_msg_port->mp_SigBit);
    while (NULL != GetMsg(game_msg_port))
        {
        timeout=FALSE;
        switch(game_event.ie_Code)
            {
            case IECODE_LBUTTON:
                printf(" FIRE BUTTON PRESSED \n");
                break;

            case (IECODE_LBUTTON | IECODE_UP_PREFIX):
                printf(" FIRE BUTTON RELEASED \n");
                if (3 == ++button_count)
                    not_finished = FALSE;
                break;

            case IECODE_RBUTTON:
                printf(" ALT BUTTON PRESSED \n");
                button_count = 0;
                break;

            case (IECODE_RBUTTON | IECODE_UP_PREFIX):
                printf(" ALT BUTTON RELEASED \n");
                button_count = 0;
                break;
            case IECODE_NOBUTTON:
                /* Check for change in position */
                timeout = check_move(&game_event);
                button_count = 0;
                break;

            default:
                break;
            }

        if (timeout)
            timeouts++;
        else
            timeouts=0;
        }
    }
}

/*-----------------------------------------------------------------------
** allocate the controller if it is available.
** you allocate the controller by setting its type to something
** other than GPCT_NOCONTROLLER.  Before you allocate the thing
** you need to check if anyone else is using it (it is free if
** it is set to GPCT_NOCONTROLLER).
*/
BOOL set_controller_type(BYTE type, struct IOStdReq *game_io_msg)
{
BOOL success = FALSE;
BYTE controller_type = 0;

/* begin critical section
** we need to be sure that between the time we check that the controller
** is available and the time we allocate it, no one else steals it.
*/
Forbid();

game_io_msg->io_Command = GPD_ASKCTYPE;    /* inquire current status */
game_io_msg->io_Flags   = IOF_QUICK;
game_io_msg->io_Data    = (APTR)&controller_type; /* put answer in here */
game_io_msg->io_Length  = 1;
DoIO(game_io_msg);

/* No one is using this device unit, let's claim it */
if (controller_type == GPCT_NOCONTROLLER)
    {
    game_io_msg->io_Command = GPD_SETCTYPE;
    game_io_msg->io_Flags   = IOF_QUICK;
    game_io_msg->io_Data    = (APTR)&type;
    game_io_msg->io_Length  = 1;
    DoIO( game_io_msg);
    success = TRUE;
    }

Permit(); /* critical section end */
return(success);
}

/*-----------------------------------------------------------------------
** tell the gameport when to trigger.
*/
VOID set_trigger_conditions(struct GamePortTrigger *gpt,
                            struct IOStdReq *game_io_msg)
{
/* trigger on all joystick key transitions */
gpt->gpt_Keys    = GPTF_UPKEYS | GPTF_DOWNKEYS;
gpt->gpt_XDelta = JOY_X_DELTA;
gpt->gpt_YDelta = JOY_Y_DELTA;
/* timeout trigger every TIMEOUT_SECONDS second(s) */
gpt->gpt_Timeout = (UWORD)(SysBase->VBlankFrequency) * TIMEOUT_SECONDS;

game_io_msg->io_Command = GPD_SETTRIGGER;
game_io_msg->io_Flags   = IOF_QUICK;
game_io_msg->io_Data    = (APTR)gpt;
game_io_msg->io_Length  = (LONG)sizeof(struct GamePortTrigger);
DoIO(game_io_msg);
}
```

```
/*----------------------------------------------------------------------
** clear the buffer.  do this before you begin to be sure you
** start in a known state.
*/
VOID flush_buffer(struct IOStdReq *game_io_msg)
{
game_io_msg->io_Command = CMD_CLEAR;
game_io_msg->io_Flags   = IOF_QUICK;
game_io_msg->io_Data    = NULL;
game_io_msg->io_Length  = 0;
DoIO(game_io_msg);
}

/*----------------------------------------------------------------------
** free the unit by setting its type back to GPCT_NOCONTROLLER.
*/
VOID free_gp_unit(struct IOStdReq *game_io_msg)
{
BYTE type = GPCT_NOCONTROLLER;

game_io_msg->io_Command = GPD_SETCTYPE;
game_io_msg->io_Flags   = IOF_QUICK;
game_io_msg->io_Data    = (APTR)&type;
game_io_msg->io_Length  = 1;
DoIO(game_io_msg);
}

/*----------------------------------------------------------------------
** allocate everything and go.  On failure, free any resources that
** have been allocated.  this program fails quietly--no error messages.
*/
VOID main(int argc,char **argv)
{
struct GamePortTrigger   joytrigger;
struct IOStdReq         *game_io_msg;
struct MsgPort          *game_msg_port;

/* Create port for gameport device communications */
if (game_msg_port = CreatePort("RKM_game_port",0))
    {
    /* Create message block for device IO */
    if (game_io_msg = (struct IOStdReq *)
                      CreateExtIO(game_msg_port,sizeof(*game_io_msg)))
        {
        game_io_msg->io_Message.mn_Node.ln_Type = NT_UNKNOWN;

        /* Open the right/back (unit 1, number 2) gameport.device unit */
        if (!OpenDevice("gameport.device",1,game_io_msg,0))
            {
            /* Set controller type to joystick */
            if (set_controller_type(GPCT_ABSJOYSTICK,game_io_msg))
                {
                /* Specify the trigger conditions */
                set_trigger_conditions(&joytrigger,game_io_msg);

                printInstructions();

                /* Clear device buffer to start from a known state.
                ** There might still be events left
                */
                flush_buffer(game_io_msg);

                processEvents(game_io_msg,game_msg_port);

                /* Free gameport unit so other applications can use it ! */
                free_gp_unit(game_io_msg);
                }
            CloseDevice(game_io_msg);
            }
        DeleteExtIO(game_io_msg);
        }
    DeletePort(game_msg_port);
    }
}
```

Additional Information on the Gameport Device

Additional programming information on the gameport device can be found in the include files and the Autodocs for the gameport and input devices. Both are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Gameport Device Information	
INCLUDES	devices/gameport.h
	devices/gameport.i
	devices/inputevent.h
	devices/inputevent.i
AUTODOCS	gameport.doc

chapter six

INPUT DEVICE

The input device is the central collection point for input events disseminated throughout the system. The best way to describe the input device is a manager of a stream with feeders. The input device itself and other modules such as the file system add events to the stream; so do input device "users"—programs or other devices that use parts of the stream or change it in some way. Feeders of the input device include the keyboard, timer and gameport devices. The keyboard, gameport, and timer devices are special cases in that the input device opens them and asks them for input. Users of the input device include Intuition and the console device.

New Features for Version 2.0

Feature	Description
IECLASS_NEWPOINTERPOS	Input Event Class
IECLASS_MENUHELP	Input Event Class
IECLASS_CHANGEWINDOW	Input Event Class
IESUBCLASS_COMPATIBLE	Input Event SubClass
IESUBCLASS_PIXEL	Input Event SubClass
IESUBCLASS_TABLET	Input Event SubClass
PeekQualifier()	Function

Compatibility Warning: The new features for the 2.0 input device are not backwards compatible.

Input Device Commands and Functions

Command	Operation
CMD_FLUSH	Purge all active and queued requests for the input device.
CMD_RESET	Reset the input port to its initialized state. All active and queued I/O requests will be aborted. Restarts the device if it has been stopped.
CMD_START	Restart the currently active input (if any) and resume queued I/O requests.
CMD_STOP	Stop any currently active input and prevent queued I/O requests from starting.
IND_ADDHANDLER	Add an input-stream handler into the handler chain.
IND_REMHANDLER	Remove an input-stream handler from the handler chain.
IND_SETMPORT	Set the controller port to which the mouse is connected.
IND_SETMTRIG	Set conditions that must be met by a mouse before a pending read request will be satisfied.
IND_SETMTYPE	Set the type of device at the mouse port.
IND_SETPERIOD	Set the period at which a repeating key repeats.
IND_SETTHRESH	Set the repeating key hold-down time before repeat starts.
IND_WRITEEVENT	Propagate an input event stream to all devices.

Input Device Function

PeekQualifier()	Return the input device's current qualifiers. (V36)

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the input device.
CheckIO()	Return the status of an I/O request.
CloseDevice()	Relinquish use of the input device.
DoIO()	Initiate a command and wait for completion (synchronous request).
OpenDevice()	Obtain shared use of the input device.
SendIO()	Initiate a command and return immediately (asynchronous request).

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure of type **IOStdReq**. This structure will be used to communicate commands to the input device.
CreatePort()	Create a signal message port for reply messages from the input device. Exec will signal a task when a message arrives at the reply port.
DeleteExtIO()	Delete an I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The input device operates like the other Amiga devices. To use it, you must first open the input device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

A number of structures are used by the input device to do its processing. Some are used to pass commands and data to the device, some are used to describe input events like mouse movements and key depressions, and one structure is used to describe the environment for input event handlers.

The I/O request used by the input device for most commands is **IOStdReq**.

```
struct IOStdReq
{
    struct Message io_Message; /* message reply port */
    struct Device  *io_Device; /* device node pointer */
    struct Unit    *io_Unit;   /* unit */
    UWORD  io_Command;         /* input device command */
    UBYTE  io_Flags;           /* input device flags */
    BYTE   io_Error;           /* error code */
    ULONG  io_Length;          /* number of bytes to transfer */
    APTR   io_Data;            /* pointer to data area */
};
```

See the include file *exec/io.h* for the complete structure definition.

Two of the input device commands—IND_SETTHRESH and IND_SETPERIOD—require a time specification and must use a **timerequest** structure instead of an **IOStdReq**.

```
struct timerequest
{
    struct IORequest tr_node;
    struct timeval tr_time;
};
```

As you can see, the **timerequest** structure includes an **IORequest** structure. The **io_Command** field of the**IORequest** indicates the command to the input device and the **timeval** structure sets the time values. See the include file *devices/timer.h* for the complete structure definition.

> *In Case You Feel Like Reinventing the Wheel...* You could define a "super-IORequest" structure for the input device which would combine the **IOStdReq** fields with the **timeval** structure of the **timerequest** structure.

OPENING THE INPUT DEVICE

Three primary steps are required to open the input device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an I/O request structure of type **IOStdReq** or **timerequest**. The I/O request created by the **CreateExtIO()** function will be used to pass commands and data to the input device.
- Open the Input device. Call **OpenDevice()**, passing the I/O request.

```
struct MsgPort  *InputMP;    /* Message port pointer */
struct IOStdReq *InputIO;    /* I/O request pointer */

if (InputMP=CreatePort(0,0) )
    if (InputIO=(struct IOStdReq *)
                CreateExtIO(InputMP,sizeof(struct IOStdReq)) )
        if (OpenDevice("input.device",0L,(struct IORequest *)InputIO,0) )
            printf("input.device did not open\n");
```

The above code will work for all the input device commands except for the ones which require a time specification. For those, the code would look like this:

```
#include <devices/timer.h>

struct MsgPort  *InputMP;       /* Message port pointer */
struct timerequest *InputIO;    /* I/O request pointer */

if (InputMP=CreatePort(0,0) )
    if (InputIO=(struct timerequest *)
                CreateExtIO(InputMP,sizeof(struct timerequest)) )
        if (OpenDevice("input.device",0L,(struct IORequest *)InputIO,0) )
            printf("input.device did not open\n");
```

INPUT DEVICE EVENT TYPES

The input device is automatically opened by the console device when the system boots. When the input device is opened, a task named "input.device" is started. The input device task communicates directly with the keyboard device to obtain raw key events. It also communicates with the gameport device to obtain mouse button and mouse movement events and with the timer device to obtain time events. In addition to these events, you can add your own input events to the input device, to be fed to the handler chain (see below).

The keyboard device is accessible directly (see the "Keyboard Device" chapter). However, once the input.device task has started, you should not read events from the keyboard device directly, since doing so will deprive the input device of the events and confuse key repeating.

The gameport device has two units. As you view the Amiga, looking at the gameport connectors, the left connector is assigned as the primary mouse input for Intuition and contributes gameport input events to the input event stream.

The right connector is handled by the other gameport unit and is currently unassigned. While the input device task is running, that task expects to read the input from the left connector. Direct use of the gameport device is covered in the "Gameport Device" chapter of this manual.

The timer device is used to generate time events for the input device. It is also used to control key repeat rate and key repeat threshold. The timer device is a shared-access device and is described in "Timer Device" chapter of this manual.

The device-specific commands are described below. First though, it may be helpful to consider the types of input events that the input device deals with. An input event is a data structure that describes the following:

- The class of the event — often describes the device that generated the event.
- The subclass of the event — space for more information if needed.
- The code — keycode if keyboard, button information if mouse, others.
- A qualifier such as "Alt key also down,"or "key repeat active".

- A position field that contains a data address or a mouse position count.
- A time stamp, to determine the sequence in which the events occurred.
- A link-field by which input events are linked together.
- The class, subclass, code and qualifier of the previous down key.

The full definitions for each field can be found in the include file *devices/inputevent.h*. You can find more information about input events in the "Gameport Device" and "Console Device" chapters of this manual.

The various types of input events are listed below.

Input Device Event Types

IECLASS_NULL	A NOP input event
IECLASS_RAWKEY	A raw keycode from the keyboard device
IECLASS_RAWMOUSE	The raw mouse report from the gameport device
IECLASS_EVENT	A private console event
IECLASS_POINTERPOS	A pointer position report
IECLASS_TIMER	A timer event
IECLASS_GADGETDOWN	Select button pressed down over a gadget (address in **ie_EventAddress**)
IECLASS_GADGETUP	Select button released over the same gadget (address in **ie_EventAddress**)
IECLASS_REQUESTER	Some requester activity has taken place.
IECLASS_MENULIST	This is a menu number transmission (menu number is in **ie_Code**)
IECLASS_CLOSEWINDOW	User has selected the active window's Close Gadget
IECLASS_SIZEWINDOW	This window has a new size
IECLASS_REFRESHWINDOW	The window pointed to by **ie_EventAddress** needs to be refreshed
IECLASS_NEWPREFS	New preferences are available
IECLASS_DISKREMOVED	The disk has been removed
IECLASS_DISKINSERTED	The disk has been inserted
IECLASS_ACTIVEWINDOW	The window is about to be been made active
IECLASS_INACTIVEWINDOW	The window is about to be made inactive
IECLASS_NEWPOINTERPOS	Extended-function pointer position report (V36)
IECLASS_MENUHELP	Help key report during Menu session (V36)
IECLASS_CHANGEWINDOW	The Window has been modified with move, size, zoom, or change (V36)

There is a difference between simply receiving an input event from a device and actually becoming a handler of an input event stream. A handler is a routine that is passed an input event list. It is up to the handler to decide if it can process the input events. If the handler does not recognize an event, it leaves it undisturbed in the event list.

It All Flows Downhill. Handlers can themselves generate new linked lists of events which can be passed down to lower priority handlers.

The **InputEvent** structure is used by the input device to describe an input event such as a keypress or a mouse movement.

```
struct InputEvent
{
    struct  InputEvent *ie_NextEvent;   /* the chronologically next event */
    UBYTE   ie_Class;                   /* the input event class */
    UBYTE   ie_SubClass;                /* optional subclass of the class */
    UWORD   ie_Code;                    /* the input event code */
    UWORD   ie_Qualifier;               /* qualifiers in effect for the event*/
    union
    {
        struct
        {
            WORD    ie_x;               /* the pointer position for the event*/
            WORD    ie_y;
        } ie_xy;
        APTR    ie_addr;                /* the event address */
        struct
        {
            UBYTE   ie_prev1DownCode;   /* previous down keys for dead */
            UBYTE   ie_prev1DownQual;   /*   key translation: the ie_Code */
            UBYTE   ie_prev2DownCode;   /*   & low byte of ie_Qualifier for */
            UBYTE   ie_prev2DownQual;   /*   last & second last down keys */
        } ie_dead;
    } ie_position;
    struct timeval ie_TimeStamp;        /* the system tick at the event */
};
```

The **IEPointerPixel** and **IEPointerTablet** structures are used to set the mouse position with the IECLASS_NEWPOINTERPOS input event class.

```
struct IEPointerPixel
{
    struct Screen       *iepp_Screen;   /* pointer to an open screen */
    struct
    {                           /* pixel coordinates in iepp_Screen */
        WORD    X;
        WORD    Y;
    } iepp_Position;
};
struct IEPointerTablet
{
    struct
    {
        UWORD   X;
        UWORD   Y;
    } iept_Range;       /* 0 is min, these are max      */
    struct
    {
        UWORD   X;
        UWORD   Y;
    } iept_Value;       /* between 0 and iept_Range     */

    WORD iept_Pressure; /* -128 to 127 (unused, set to 0)   */
};
```

See the include file *devices/inputevent.h* for the complete structure definitions.

For input device handler installation, the **Interrupt** structure is used.

```
struct Interrupt
{
    struct Node is_Node;
    APTR   is_Data;         /* server data segment */
    VOID   (*is_Code)();    /* server code entry   */
};
```

See the include file *exec/interrupts.h* for the complete structure definition.

CLOSING THE INPUT DEVICE

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**. All I/O requests must be complete before **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()**:

```
if (!(CheckIO(InputIO)))
    {
    AbortIO(InputIO);  /* Ask device to abort request, if pending */
    }
WaitIO(InputIO);    /* Wait for abort, then clean up */
CloseDevice((struct IORequest *)InputIO);
```

Using the Mouse Port With the Input Device

To get mouse port information you must first set the current mouse port by passing an **IOStdReq** to the device with IND_SETMPORT set in **io_Command** and a pointer to a byte set in **io_Data**. If the byte is set to 0 the left controller port will be used as the current mouse port; if it is set to 1, the right controller port will be used.

```
BYTE port = 1;      /* set mouse port to right controller */

InputIO->io_Data = &port;
InputIO->io_Flags = IOF_QUICK;
InputIO->io_Length = 1;
InputIO->io_Command = IND_SETMPORT;
BeginIO((struct IORequest *)InputIO);
if (InputIO->io_Error)
    printf("\nSETMPORT failed %d\n",InputIO->io_Error);
```

> *Put That Back!* The default mouse port is the left controller. Don't forget to set the mouse port back to the left controller before exiting if you change it to the right controller during your application.

SETTING THE CONDITIONS FOR A MOUSE PORT REPORT

You set the conditions for a mouse port report by passing an **IOStdReq** to the device with IND_SETMTRIG set in **io_Command**, the address of a **GamePortTrigger** structure set in **io_Data** and the length of the structure set in **io_Length**.

```
struct GamePortTrigger InputTR;

InputIO->io_Data = (APTR)InputTR;     /* set trigger conditions */
InputIO->io_Command = IND_SETMTRIG;   /* from InputTR */
InputIO->io_Length = sizeof(struct GamePortTrigger);
DoIO(InputIO);
```

The information needed for mouse port report setting is contained in a **GamePortTrigger** data structure which is defined in the include file *devices/gameport.h.*

```
struct GamePortTrigger
{
    UWORD    gpt_Keys;       /* key transition triggers */
    UWORD    gpt_Timeout;    /* time trigger (vertical blank units) */
    UWORD    gpt_XDelta;     /* X distance trigger */
    UWORD    gpt_YDelta;     /* Y distance trigger */
};
```

See the "Gameport Device" chapter of this manual for a full description of setting mouse port trigger conditions.

Adding an Input Handler

You add an input-stream handler to the input chain by passing an **IOStdReq** to the device with IND_ADDHANDLER set in **io_Command** and a pointer to an **Interrupt** structure set in **io_Data**.

```
struct Interrupt *InputHandler;
struct IOStdReq  *InputIO

InputHandler->is_Code=ButtonSwap;         /* Address of code */
InputHandler->is_Data=NULL;               /* User Value passed in A1 */
InputHandler->is_Node.ln_Pri=100;         /* Priority in food chain */
InputHandler->is_Node.ln_Name=NameString; /* Name of handler */

InputIO->io_Data=(APTR)inputHandler;      /* Point to the structure */
InputIO->io_Command=IND_ADDHANDLER;       /* Set command ... */
DoIO((struct IORequest *)InputIO);       /* DoIO( ) the command */
```

Intuition is one of the input device handlers and normally distributes most of the input events.

Intuition inserts itself at priority position 50. The console device sits at priority position 0. You can choose the position in the chain at which your handler will be inserted by setting the priority field in the list-node part of the interrupt data structure you pass to this routine.

> *Speed Saves.* *Any* processing time expended by a handler subtracts from the time available before the next event happens. Therefore, handlers for the input stream *must* be fast. For this reason it is recommended that the handlers be written in assembly.

RULES FOR INPUT DEVICE HANDLERS

The following rules should be followed when you are designing an input handler:

- If an input handler is capable of processing a specific kind of an input event and that event has no links (**ie_NextEvent** = 0), the handler can end the handler chain by returning a NULL (0) value.
- If there are multiple events linked together, the handler is free to unlink an event from the input event chain, thereby passing a shorter list of events to subsequent handlers. The starting address of the modified list is the return value.
- If a handler wishes to add new events to the chain that it passes to a lower-priority handler, it may initialize memory to contain the new event or event chain. The handler, when it again gets control on the next round of event handling, should assume nothing about the current contents of the memory blocks attached to the event chain. Lower priority handlers may have modified the memory as they handled their part of the event. The handler that allocates the memory for this purpose should keep track of the starting address and the size of this memory chunk so that the memory can be returned to the free memory list when it is no longer needed.

Your assembly language handler routine should be structured similar to the following pseudo-language statement:

```
newEventChain = yourHandlerCode(oldEventChain, yourHandlerData);
     d0       =                       a0             a1
```

where:

- **yourHandlerCode** is the entry point to your routine.
- **oldEventChain** is the starting address for the current chain of input events.
- **yourHandlerData** is a user-definable value, usually a pointer to some data structure your handler requires.
- **newEventChain** is the starting address of an event chain which you are passing to the next handler, if any.

When your handler code is called, the event chain is passed in A0 and the handler data is passed in A1. (You may choose not to use A1.) When your code returns, it should return the pointer to the event chain in D0. If all of the events were removed by the routine, return NULL. A NULL (0) value terminates the handling thus freeing more CPU resources.

Memory that you use to describe a new input event that you have added to the event chain is available for reuse or deallocation when the handler is called again or after the IND_REMHANDLER command for the handler is complete. There is no guarantee that any field in the event is unchanged since a handler may change any field of an event that comes through the food chain.

> *Do Not Confuse the Device.* Altering a repeat key report will confuse the input device when it tries to stop the repeating after the key is raised under pre-V36 Kickstart.

Because IND_ADDHANDLER installs a handler in any position in the handler chain, it can, for example, ignore specific types of input events as well as act upon and modify existing streams of input. It can even create new input events for Intuition or other programs to interpret.

REMOVING AN INPUT HANDLER

You remove a handler from the handler chain by passing an **IOStdReq** to the device IND_REMHANDLER set in **io_Command** and a pointer to the **Interrupt** structure used to add the handler.

```
struct Interrupt *InputHandler;
struct IOStdReq  *InputIO;

InputIO->io_Data=(APTR)InputHandler;    /* Which handler to REM */
InputIO->io_Command=IND_REMHANDLER;     /* The REM command */
DoIO((struct IORequest *)InputIO);     /* Send the command */
```

Writing Events to the Input Device Stream

Typically, input events are internally generated by the timer device, keyboard device, and input device.

An application can also generate an input event by setting the appropriate fields for the event in an **InputEvent** structure and sending it to the input device. It will then be treated as any other event and passed through to the input handler chain. However, I/O requests for IND_WRITEVENT cannot be made from interrupt code.

You generate an input event by passing an **IOStdReq** to the device with IND_WRITEEVENT set in **io_Command**, a pointer to an **InputEvent** structure set in **io_Data** and the length of the structure set in **io_Length**.

```
struct InputEvent *FakeEvent;
struct IOStdReq   *InputIO;

InputIO->io_Data=(APTR)FakeEvent;
InputIO->io_Length=sizeof(struct InputEvent);
InputIO->io_Command=IND_WRITEEVENT;
DoIO((struct IORequest *)InputIO);
```

You Know What Happens When You Assume. This command propagates the input event through the handler chain. The handlers may link other events onto the end of this event or modify the contents of the data structure you constructed in any way they wish. Therefore, do not assume any of the data will be the same from event to event.

SETTING THE POSITION OF THE MOUSE

One use of writing input events to the input device is to set the position of the mouse pointer. The mouse pointer can be positioned by using the input classes IECLASS_POINTERPOS and IECLASS_NEWPOINTERPOS.

There are two ways to set the position of the mouse pointer using the pre-V36 Kickstart input class IECLASS_POINTERPOS:

- At an absolute position on the current screen.
- At a position relative to the current mouse pointer position on the current screen.

In both cases, you set the **Class** field of the **InputEvent** structure to IECLASS_POINTERPOS, **ie_X** with the new x-coordinate and **ie_Y** with the new y-coordinate. Absolute positioning is done by setting **ie_Qualifier** to NULL and relative positioning is done by setting **ie_Qualifier** to RELATIVE_MOUSE.

Once the proper values are set, pass an **IOStdReq** to the input device with a pointer to the **InputEvent** structure set in **io_Data** and **io_Command** set to IND_WRITEEVENT.

There are three ways to set the mouse pointer position using IECLASS_NEWPOINTERPOS:

- At an absolute x-y coordinate on a screen—you specify the exact location of the pointer and which screen.
- At an relative x-y coordinate—you specify where it will go in relation to the current pointer position and which screen.
- At a normalized position on a tablet device—you specify the maximum x-value and y-value of the tablet and an x-y coordinate between them and the input device will normalize it to fit.

The basic steps required are the same for all three methods.

- Get a pointer to the screen where you want to position the pointer. This is not necessary for the tablet device.
- Set up a structure to indicate the new position of the pointer.

For absolute and relative positioning, you set up an **IEPointerPixel** structure with **iepp_Position.X** set to the new x-coordinate, **iepp_Position.Y** set to the new y-coordinate and **iepp_Screen** set to the screen pointer. You set up an **InputEvent** structure with **ie_SubClass** set to IESUBCLASS_PIXEL, a pointer to the **IEPointerPixel** structure set in **ie_EventAddress**, IECLASS_NEWPOINTERPOS set in **Class**, and **ie_Qualifier** set to either IEQUALIFIER_RELATIVEMOUSE for relative positioning or NULL for absolute positioning.

For tablet positioning, you set up an **IEPointerTablet** structure with **iept_Range.X** set to the maximum x-coordinate and **iept_Range.Y** set to the maximum y-coordinate, and **iept_Value.X** set to the new x-coordinate and **iept_Value.Y** set to the new y-coordinate. You set up an **InputEvent** structure with a pointer to the **IEPointerTablet** structure set in **ie_EventAddress**, **ie_SubClass** to IESUBCLASS_TABLET and **Class** set to IECLASS_NEWPOINTERPOS.

Finally, for all three methods, pass an **IOStdReq** to the device with a pointer to the **InputEvent** structure set in **io_Data** and **io_Command** set to IND_WRITEEVENT.

The following example sets the mouse pointer at an absolute position on a public screen using IECLASS_NEWPOINTERPOS. Notice that it uses V36 functions wherever possible.

```
/*
 * Set_Mouse.c
 *
 * This example sets the mouse at x=100 and y=200
 *
 * Compile with SAS C 5.10: LC -b1 -cfistq -v -y -L
 *
 * Requires Kickstart 36 or greater.
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <devices/input.h>
#include <devices/inputevent.h>
#include <intuition/screens.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct IntuitionBase *IntuitionBase;

void main(void)
{
struct IOStdReq    *InputIO;          /* I/O request block */
struct MsgPort     *InputMP;          /* Message port */
struct InputEvent *FakeEvent;         /* InputEvent pointer */
struct IEPointerPixel *NewPixel;      /* New mouse position pointer */
struct Screen *PubScreen;             /* Screen pointer */

if (InputMP = CreateMsgPort())
    {
    if ((FakeEvent = AllocMem(sizeof(struct InputEvent),MEMF_PUBLIC)) &&
        (NewPixel  = AllocMem(sizeof(struct IEPointerPixel),MEMF_PUBLIC)) )
        {
        if (InputIO = CreateIORequest(InputMP,sizeof(struct IOStdReq)))
            {
            if (!OpenDevice("input.device",NULL,(struct IORequest *)InputIO,NULL))
                {
                    /* Open Intuition library */
                if (IntuitionBase = (struct IntuitionBase *)
                                     OpenLibrary("intuition.library",36L))
                    {
                    /* Get pointer to screen and lock screen */
                    if (PubScreen = (struct Screen *)LockPubScreen(NULL))
                        {
                        /* Set up IEPointerPixel fields */
                        NewPixel->iepp_Screen = (struct Screen *)PubScreen;  /* WB screen */
                        NewPixel->iepp_Position.X = 100;  /* put pointer at x = 100 */
                        NewPixel->iepp_Position.Y = 200;  /* put pointer at y = 200 */

                        /* Set up InputEvent fields */
                        FakeEvent->ie_EventAddress = (APTR)NewPixel; /* IEPointerPixel */
                        FakeEvent->ie_NextEvent = NULL;
                        FakeEvent->ie_Class = IECLASS_NEWPOINTERPOS;  /* new mouse pos */
                        FakeEvent->ie_SubClass = IESUBCLASS_PIXEL;    /* on pixel */
                        FakeEvent->ie_Code = IECODE_NOBUTTON;
                        FakeEvent->ie_Qualifier = NULL;    /* absolute positioning */

                        InputIO->io_Data = (APTR)FakeEvent;    /* InputEvent */
                        InputIO->io_Length = sizeof(struct InputEvent);
                        InputIO->io_Command = IND_WRITEEVENT;
                        DoIO((struct IORequest *)InputIO);

                        /* Unlock screen */
                        UnlockPubScreen(NULL,PubScreen);
                        }
                    else
                        printf("Could not get pointer to screen\n");

                    /* Close intuition library */
                    CloseLibrary(IntuitionBase);
                    }
                else
                    printf("Error: Could not open V36 or higher intuition.library\n");

                CloseDevice((struct IORequest *)InputIO);
                }
            else
                printf("Error: Could not open input.device\n");

            DeleteIORequest(InputIO);
            }
        else
            printf("Error: Could not create I/O request\n");

        FreeMem(FakeEvent,sizeof(struct InputEvent));
        FreeMem(NewPixel,sizeof(struct IEPointerPixel));
        }
    else
        printf("Error: Could not allocate memory for structures\n");

    DeleteMsgPort(InputMP);
    }
else
    printf("Error: Could not create message port\n");
}
```

Setting the Key Repeat Threshold

The key repeat threshold is the number of seconds and microseconds a user must hold down a key before it begins to repeat. This delay is normally set by the Preferences tool or by Intuition when it notices that the Preferences have been changed, but you can also do it directly through the input device.

You set the key repeat threshold by passing a **timerequest** with IND_SETTHRESH set in **io_Command** and the number of seconds to delay set in **tv_secs** and the number of microseconds to delay set in **tv_micro**.

```
#include <devices/timer.h>

struct timerequest *InputTime;  /* Initialize with CreateExtIO() before using */

InputTime->tr_time.tv_secs=1;         /* 1 second */
InputTime->tr_time.tv_micro=500000;   /* 500000 microseconds */
InputTime->tr_node.io_Command=IND_SETTHRESH;
DoIO((struct IORequest *)InputTime);
```

The code above will set the key repeat threshold to 1.5 seconds.

Setting the Key Repeat Interval

The key repeat interval is the time period, in seconds and microseconds, between key repeat events once the initial key repeat threshold has elapsed. (See "Setting the Key Repeat Threshold" above.) Like the key repeat threshold, this is normally issued by Intuition and preset by the Preferences tool.

You set the key repeat interval by passing a **timerequest** with IND_SETPERIOD set in **io_Command** and the number of seconds set in **tv_secs** and the number of microseconds set in **tv_micro**.

```
struct timerequest *InputTime; /* Initialize with CreateExtIO() before using */

InputTime->tr_time.tv_secs=0;
InputTime->tr_time.tv_micro=12000;  /* .012 seconds */
InputTime->tr_node.io_Command=IND_SETPERIOD;
DoIO((struct IORequest *)InputTime);
```

The code above sets the key repeat interval to .012 seconds.

> *The Right Tool For The Right Job.* As previously stated, you *must* use a **timerequest** structure with IND_SETTHRESH and IND_SETPERIOD.

Determining the Current Qualifiers

Some applications need to know whether the user is holding down a qualifier key or a mouse button during an operation. To determine the current qualifiers, you call the input device function **PeekQualifier()**.

PeekQualifier() returns what the input device *considers* to be the current qualifiers at the time **PeekQualifier()** is called (e.g., keyboard qualifiers and mouse buttons). This does not include any qualifiers which have been added, removed or otherwise modified by input handlers.

In order to call the function, you must set a pointer to the input device base address. The pointer must be declared in the global data area of your program. Once you set the pointer, you can call the function. *You must open the device in order to access the device base address.*

PeekQualifier() returns an unsigned word with bits set according to the qualifiers in effect at the *time* the function is called. It takes no parameters.

```
struct Library *InputBase;      /* Input device base address pointer */

VOID main(VOID)
{
struct IOStdReq   *InputIO;          /* I/O request block */
UWORD  Quals;                        /* qualifiers */
   .
   .
   .
if (!OpenDevice("input.device",NULL,(struct IORequest *)InputIO,NULL))
     {
     /* Set input device base address in InputBase */
     InputBase = (struct Library *)InputIO->io_Device;

     /* Call the function */
     Quals = PeekQualifier();
     .
     .
     .
     CloseDevice(InputIO);
     }
}
```

The qualifiers returned are listed in the table below.

Bit	Qualifier	Key or Button
0	IEQUALIFIER_LSHIFT	Left Shift
1	IEQUALIFIER_RSHIFT	Right Shift
2	IEQUALIFIER_CAPSLOCK	Caps Lock
3	IEQUALIFIER_CONTROL	Control
4	IEQUALIFIER_LALT	Left Alt
5	IEQUALIFIER_RALT	Right Alt
6	IEQUALIFIER_LCOMMAND	Left-Amiga
7	IEQUALIFIER_RCOMMAND	Right-Amiga
12	IEQUALIFIER_MIDBUTTON	Middle Mouse
13	IEQUALIFIER_RBUTTON	Right Mouse
14	IEQUALIFIER_LEFTBUTTON	Left Mouse

Input Device and Intuition

There are several ways to receive information from the various devices that are part of the input device. The first way is to communicate directly with the device. This method is not recommended while the input device task is running – which is most of the time. The second way is to become a handler for the stream of events which the input device produces. That method is shown above.

The third method of getting input from the input device is to retrieve the data from the console device or from the IDCMP (Intuition Direct Communications Message Port). These are the preferred methods for applications in a multitasking environment because each application can receive juts its own input (i.e., only the input which occurs when one of its window is active). See the "Intuition" chapter of *Amiga ROM Kernel Reference Manual: Libraries* for more information on IDCMP messages. See the "Console Device" chapter of this manual for more information on console device I/O.

Example Input Device Program

```
/*
 * Swap_Buttons.c
 *
 * This example swaps the function of the left and right mouse buttons
 * The C code is just the wrapper that installs and removes the
 * input.device handler that does the work.
 *
 * The handler is written in assembly code since it is important that
 * handlers be as fast as possible while processing the input events.
 *
 * Compile and link as follows:
 *
 * SAS C 5.10:
 *  LC -b1 -cfirst -v -w Swap_Buttons.c
 *
 * Adapt assemble:
 *  HX68 InputHandler.a to InputHandler.o
 *
 * BLink:
 *  BLink from LIB:c.o+Swap_Buttons.o+InputHandler.o LIB LIB:lc.lib LIB:amiga.lib TO Swap_Buttons
 *
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/interrupts.h>
#include <devices/input.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

UBYTE NameString[]="Swap Buttons";

struct NewWindow mywin={50,40,124,18,0,1,CLOSEWINDOW,
                        WINDOWDRAG|WINDOWCLOSE|SIMPLE_REFRESH|NOCAREREFRESH,
                        NULL,NULL,NameString,NULL,NULL,0,0,0,0,WBENCHSCREEN};

extern VOID ButtonSwap();

extern struct IntuitionBase *IntuitionBase;

/*
 * This routine opens a window and waits for the one event that
 * can happen (CLOSEWINDOW)  This is just to let the user play with
 * the swapped buttons and then close the program...
 */
VOID WaitForUser(VOID)
{
struct Window  *win;

if (IntuitionBase=(struct IntuitionBase *)
                                OpenLibrary("intuition.library",33L))
    {
    if (win=OpenWindow(&mywin))
        {
        WaitPort(win->UserPort);
        ReplyMsg(GetMsg(win->UserPort));

        CloseWindow(win);
        }
    CloseLibrary((struct Library *)IntuitionBase);
    }
}
```

```
VOID main(VOID)
{
struct IOStdReq  *inputReqBlk;
struct MsgPort   *inputPort;
struct Interrupt *inputHandler;

if (inputPort=CreatePort(NULL,NULL))
    {
    if (inputHandler=AllocMem(sizeof(struct Interrupt),
                               MEMF_PUBLIC|MEMF_CLEAR))
        {
        if (inputReqBlk=(struct IOStdReq *)CreateExtIO(inputPort,
                                                  sizeof(struct IOStdReq)))
            {
            if (!OpenDevice("input.device",NULL,
                             (struct IORequest *)inputReqBlk,NULL))
                {
                inputHandler->is_Code=ButtonSwap;
                inputHandler->is_Data=NULL;
                inputHandler->is_Node.ln_Pri=100;
                inputHandler->is_Node.ln_Name=NameString;
                inputReqBlk->io_Data=(APTR)inputHandler;
                inputReqBlk->io_Command=IND_ADDHANDLER;
                DoIO((struct IORequest *)inputReqBlk);

                WaitForUser();

                inputReqBlk->io_Data=(APTR)inputHandler;
                inputReqBlk->io_Command=IND_REMHANDLER;
                DoIO((struct IORequest *)inputReqBlk);

                CloseDevice((struct IORequest *)inputReqBlk);
                }
            else
                printf("Error: Could not open input.device\n");

            DeleteExtIO((struct IORequest *)inputReqBlk);
            }
        else
            printf("Error: Could not create I/O request\n");

        FreeMem(inputHandler,sizeof(struct Interrupt));
        }
    else
        printf("Error: Could not allocate interrupt struct memory\n");

    DeletePort(inputPort);
    }
else
    printf("Error: Could not create message port\n");
}


***************************************************************************
*       InputHandler.a
*
* InputHandler that does a Left/Right mouse button swap...
*
* See Swap_Buttons.c for details on how to compile/assemble/link...
*
***********************************************************************
*
* Required includes...
*
        INCDIR  "include:"
        INCLUDE "exec/types.i"
        INCLUDE "exec/io.i"
        INCLUDE "devices/inputevent.i"
*
***********************************************************************
*
* Make the entry point external...
*
        xdef    _ButtonSwap
*
***********************************************************************
*
```

```
* This is the input handler that will swap the
* mouse buttons for left handed use.
*
* The event list gets passed to you in  a0.
* The is_Data field is passed to you in a1.
* This example does not use the is_Data field...
*
* On exit you must return the event list in d0.  In this way
* you could add or remove items from the event list.
*
* The handler gets called here...
*
*
_ButtonSwap:    move.l  a0,-(sp)        ; Save the event list
*
* Since the event list could be a linked list, we start a loop
* here to handle all of the events passed to us.
*
CheckLoop:      move.w  ie_Qualifier(a0),d1             ; Get qualifiers...
                move.w  d1,d0                           ; Two places...
*
* Since we are changing left and right mouse buttons, we need to make
* sure that we change the qualifiers on all of the messages.  The
* left and right mouse buttons are tracked in the message qualifiers
* for use in such things as dragging.  To make sure that we continue
* to drag correctly, we change the qualifiers.
*
CheckRight:     btst    #IEQUALIFIERB_RBUTTON,d1        ; Check for right
                beq.s   NoRight
                bset    #IEQUALIFIERB_LEFTBUTTON,d0     ; Set the left...
                beq.s   CheckLeft
NoRight:        bclr    #IEQUALIFIERB_LEFTBUTTON,d0     ; Clear the left...
*
CheckLeft:      btst    #IEQUALIFIERB_LEFTBUTTON,d1     ; Check for left
                beq.s   NoLeft
                bset    #IEQUALIFIERB_RBUTTON,d0        ; Set the right...
                beq.s   SaveQual
NoLeft:         bclr    #IEQUALIFIERB_RBUTTON,d0        ; Clear the right...
*
SaveQual:       move.w  d0,ie_Qualifier(a0)             ; Save back...
*
* The actual button up/down events are transmitted as the
* code field in RAWMOUSE events.  The code field must the be
* checked and modified when needed on RAWMOUSE events.  If the
* event is not a RAWMOUSE, we are done with it.
*
                cmp.b   #IECLASS_RAWMOUSE,ie_Class(a0)  ; Check for mouse
                bne.s   NextEvent                       ; If not, next...
*
                move.w  ie_Code(a0),d0                  ; Get code...
                move.w  d0,d1                           ; Save...
                and.w   #$7F,d0                         ; Mask UP_PREFIX
                cmp.w   #IECODE_LBUTTON,d0              ; Check for Left...
                beq.s   SwapThem                        ; If so, swap...
                cmp.w   #IECODE_RBUTTON,d0              ; Check for Right...
                bne.s   NextEvent                       ; If not, next...
*
SwapThem:       eor.w   #1,d1                           ; Flip bottom bit
                move.w  d1,ie_Code(a0)                  ; Save it...
*
* The event list is linked via a pointer to the next event
* in the first element of the structure.  That is why it is not
* nessesary to use:  move.l ie_NextEvent(a0),d0
*
* The reason I move to d0 first is that this also checks for zero.
* The last event in the list will have a NULL ie_NextEvent field.
* This is NOT as standard EXEC list where the node after the last
* node is NULL.  Input events are single-linked for performance.
*
NextEvent:      move.l  (a0),d0                         ; Get next event
                move.l  d0,a0                           ; into a0...
                bne.s   CheckLoop                       ; Do some more.
*
* All done, just return the event list...  (in d0)
*
                move.l  (sp)+,d0        ; Get event list back...
                rts                     ; return from handler...
```

Additional Information on the Input Device

Additional programming information on the input device can be found in the include files and the autodocs for the input device. Both are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Input Device Information	
INCLUDES	devices/input.h
	devices/input.i
	devices/inputevent.h
	devices/inputevent.i
AUTODOCS	input.doc

chapter seven
KEYBOARD DEVICE

The keyboard device gives low-level access to the Amiga keyboard. When you send this device the command to read one or more keystrokes from the keyboard, for each keystroke (whether key-up or key-down) the keyboard device creates a data structure called an input event to describe what happened. The keyboard device also provides the ability to do operations within the system reset processing (Ctrl-Amiga-Amiga).

Keyboard Device Commands and Functions

Command	Operation
CMD_CLEAR	Clear the keyboard input buffer. Removes any key transitions from the input buffer.
KBD_ADDRESETHANDLER	Add a reset handler function to the list of functions called by the keyboard device to clean up before a hard reset.
KBD_REMRESETHANDLER	Remove a previously added reset handler from the list of functions called by the keyboard device to clean up before a hard reset.
KBD_RESETHANDLERDONE	Indicate that a handler has completed its job and reset could possibly occur now.
KBD_READMATRIX	Read the state of every key in the keyboard. Tells the up/down state of every key.
KBD_READEVENT	Read one (or more) raw key event from the keyboard device.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the keyboard device.
AllocMem()	Allocate a block of memory.
CheckIO()	Return the status of an I/O request.
CloseDevice()	Relinquish use of the keyboard device.
DoIO()	Initiate a command and wait for it to complete (synchronous request).
FreeMem()	Free a block of previously allocated memory.
OpenDevice()	Obtain use of the keyboard device.
SendIO()	Initiate a command and return immediately (asynchronous request).
WaitIO()	Wait for the completion of an asynchronous request. When the request is complete the message will be removed from reply port.

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure. This structure will be used to communicate commands to the keyboard device.
CreatePort()	Create a signal message port for reply messages from the keyboard device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete an extended I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The keyboard device operates like the other Amiga devices. To use it, you must first open the keyboard device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The I/O request used by the keyboard device is called **IOStdReq**.

```
struct IOStdReq
{
    struct  Message io_Message;
    struct  Device  *io_Device;     /* device node pointer  */
    struct  Unit    *io_Unit;       /* unit (driver private)*/
    UWORD   io_Command;             /* device command */
    UBYTE   io_Flags;
    BYTE    io_Error;               /* error or warning num */
    ULONG   io_Actual;              /* actual number of bytes transferred */
    ULONG   io_Length;              /* requested number bytes transferred*/
    APTR    io_Data;                /* points to data area */
    ULONG   io_Offset;              /* offset for block structured devices */
};
```

See the include file *exec/io.h* for the complete structure definition.

OPENING THE KEYBOARD DEVICE

Three primary steps are required to open the keyboard device:

- Create a message port using the **CreatePort()** function.
- Create an extended I/O request structure using the **CreateExtIO()** function. **CreateExtIO()** will initialize the I/O request with your reply port.
- Open the keyboard device. Call **OpenDevice()**, passing the I/O request.

```
struct MsgPort  *KeyMP;         /* Pointer for Message Port */
struct IOStdReq *KeyIO;         /* Pointer for I/O request */

if (KeyMP=CreatePort(NULL,NULL))
    if (KeyIO=(struct IOStdReq *)
            CreateExtIO(KeyMP,sizeof(struct IOStdReq)) )
        if (OpenDevice("keyboard.device",NULL,(struct IORequest *)KeyIO,NULL))
            printf("keyboard.device did not open\n");
```

CLOSING THE KEYBOARD DEVICE

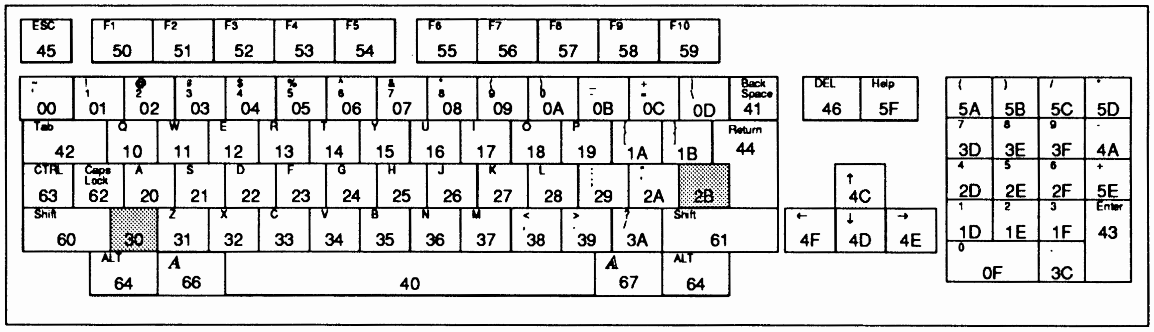
An **OpenDevice()** must eventually be matched by a call to **CloseDevice()**.

All I/O requests must be complete before **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()** and remove them with **WaitIO()**.

```
if (!(CheckIO(KeyIO)))
    {
    AbortIO(KeyIO);  /* Ask device to abort request, if pending */
    }
    WaitIO(KeyIO);   /* Wait for abort, then clean up */
CloseDevice(KeyIO);
```

Reading the Keyboard Matrix

The KBD_READMATRIX command returns the current state of every key in the key matrix (up = 0, down = 1). You provide a data area that is at least large enough to hold one bit per key, approximately 16 bytes. The keyboard layout for the A500, A2000 and A3000 is shown in the figure below, indicating the raw numeric value that each key transmits when it is pressed. This value is the numeric position that the key occupies in the key matrix.



The following example will read the key matrix and display the up-down state of all of the elements in the matrix in a table. Reading the column header and then the row number as a hex number gives you the raw key code.

```
/*
 * Read_Keyboard_Matrix.c
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/libraries.h>
#include <dos/dos.h>
#include <devices/keyboard.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }      /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }   /* really */
#endif


/*
 * There are keycodes from 0x00 to 0x7F, so the matrix needs to be
 * of 0x80 bits in size, or 0x80/8 which is 0x10 or 16 bytes...
 */
#define MATRIX_SIZE 16L

/*
 * This assembles the matrix for display that translates directly
 * to the RAW key value of the key that is up or down
 */

VOID Display_Matrix(UBYTE *keyMatrix)
{
SHORT  bitcount;
SHORT  bytecount;
SHORT   mask;
USHORT twobyte;

printf("\n    0 1 2 3 4 5 6 7");
printf("\n  +-----------------");
for (bitcount=0;bitcount<16;bitcount++)
    {
    printf("\n%x |",bitcount);
    mask=1 << bitcount;
    for (bytecount=0;bytecount<16;bytecount+=2)
        {
        twobyte=keyMatrix[bytecount] | (keyMatrix[bytecount+1] << 8);
        if (twobyte & mask)
            printf(" *");
        else
            printf(" -");
        }
    }
printf("\n\n");
}


void main(int argc, char *argv[])
{
extern struct Library *SysBase;
struct IOStdReq *KeyIO;
struct MsgPort  *KeyMP;
UBYTE    *keyMatrix;

if (KeyMP=CreatePort(NULL,NULL))
    {
    if (KeyIO=(struct IOStdReq *)CreateExtIO(KeyMP,sizeof(struct IOStdReq)))
        {
        if (!OpenDevice("keyboard.device",NULL,(struct IORequest *)KeyIO,NULL))
            {
            if (keyMatrix=AllocMem(MATRIX_SIZE,MEMF_PUBLIC|MEMF_CLEAR))
                {
                KeyIO->io_Command=KBD_READMATRIX;
                KeyIO->io_Data=(APTR)keyMatrix;
                KeyIO->io_Length= SysBase->lib_Version >= 36 ? MATRIX_SIZE : 13;
                DoIO((struct IORequest *)KeyIO);

                /* Check for CLI startup... */
                if (argc)
                    Display_Matrix(keyMatrix);

                FreeMem(keyMatrix,MATRIX_SIZE);
                }
            else
                printf("Error: Could not allocate keymatrix memory\");

            CloseDevice((struct IORequest *)KeyIO);
            }
        else
            printf("Error: Could not open keyboard.device\n");

        DeleteExtIO((struct IORequest *)KeyIO);
        }
    else
        printf("Error: Could not create I/O request\n");

    DeletePort(KeyMP);
    }
else
    printf("Error: Could not create message port\n");
}
```

In addition to the matrix data returned in **io_Data**, **io_Actual** returns the number of bytes filled in **io_Data** with key matrix data, i.e., the minimum of the supplied length and the internal key matrix size.

Value of io_Length. A value of 13 in the **io_Length** field will be sufficient for most keyboards; extended keyboards will require a larger number. However, you *must* always set this field to 13 for V34 and earlier versions of Kickstart.

To find the status of a particular key—for example, to find out if the F2 key is down—you find the bit that specifies the current state by dividing the key matrix value by 8. Since hex 51 = 81, this indicates that the bit is in byte number 10 of the matrix. Then take the same number (decimal 81) and use modulo 8 to determine which bit position within that byte represents the state of the key. This yields a value of 1. So, by reading bit position 1 of byte number 10, you determine the status of the function key F2.

Amiga Reset Handling

When a user presses the Ctrl key and both left- and right-Amiga keys simulataneously (the reset sequence), the keyboard device senses this and calls a prioritized chain of reset-handlers. These might be thought of as clean-up routines that "must" be performed before reset is allowed to occur. For example, if a disk write is in progress, the system should finish that before resetting the hardware so as not to corrupt the contents of the disk.

It is important to note that not all Amigas handle reset processing in the same way. On the A500, the reset key sequence sends a hardware reset signal and never goes through the reset handlers. Also some of the early A2000s (i.e., German keyboards with the function keys the same size as the Esc key) do not handle the reset via the reset handlers. It is thus recommended that your application not rely on the reset handler abilities of the keyboard device.

ADDING A RESET HANDLER (KBD_ADDRESETHANDLER)

The KBD_ADDRESETHANDLER command adds a custom routine to the chain of reset-handlers. Reset handlers are just like any other handler and are added to the handler list with an **Interrupt** structure. The priority field in the list node of the **Interrupt** structure establishes the sequence in which reset handlers are processed by the system. Keyboard reset handlers are currently limited to the priority values of a software interrupt, that is, values of -32, -16, 0, 16, and 32.

The **io_Data** field of the I/O request is filled in with a pointer to the **Interrupt** structure and the **io_Command** field is set to KBD_ADDRESETHANDLER. These are the only two fields you need to initialize to add a reset handler. Any return value from the command is ignored. All keyboard reset handlers are activated if time permits. Normally, a reset handler will just signal the requisite task and return. The task then does whatever processing it needs to do and notifies the system that it is done by using the KBD_RESETHANDLERDONE command described below.

> *Non-interference and speed are the keys to success.* If you add your own handler to the chain, you *must* ensure that your handler allows the rest of reset processing to occur. Reset *must* continue to function. Also, if you don't execute your reset code fast enough, the system will still reboot (about 10 seconds).

REMOVING A RESET HANDLER (KBD_REMRESETHANDLER)

This command is used to remove a keyboard reset handler from the system. You need to supply the same **Interrupt** structure to this command that you used with the KBD_ADDRESETHANDLER command.

ENDING A RESET TASK (KBD_RESETHANDLERDONE)

This command tells the system that your reset handling code has completed. If you are the last outstanding reset handler, the system will reset after this call.

> *Can't Stop, Got No Brakes.* After 10 seconds, the system will reboot, regardless of outstanding reset handlers.

Here is an example program that installs a reset handler and either waits for the reboot or for the user to close the window. If there was a reboot, the window will close and, if executed from the shell, it will display a few messages. If the user closes the window, the handler is removed and the program exits cleanly.

```
/*
 * Key_Reset.c
 *
 * This is in two parts...
 *
 * Compile this C code with SAS C 5.10:
 *      lc -b1 -cfistq -v -y Key_Reset
 *
 * Assemble the ASM code with Adapt
 *   HX68 KeyHandler.a to KeyHandler.o
 *
 * Link with:
 *      Blink FROM LIB:c.o+Key_Reset.o+KeyHandler.o TO Key_Reset LIB LIB:lc.lib LIB:amiga.lib
 */

/*
 * Keyboard device reset handler example...
 */
#include <exec/types.h>
#include <exec/io.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <devices/keyboard.h>
#include <intuition/intuition.h>
#include <exec/interrupts.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/intuition_protos.h>
#include <clib/dos_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
void main();
#endif

extern VOID ResetHandler();

UBYTE NameString[]="Reset Handler Test";

struct NewWindow mywin={0,0,178,10,0,1,CLOSEWINDOW,
                        WINDOWDRAG|WINDOWCLOSE|SIMPLE_REFRESH|NOCAREREFRESH,
                        NULL,NULL,NameString,NULL,NULL,0,0,0,0,WBENCHSCREEN};

extern struct IntuitionBase *IntuitionBase;

struct MyData
    {
    struct Task   *MyTask;
           ULONG MySignal;
    };

/*
 * This routine opens a window and waits for the one event that
 * can happen (CLOSEWINDOW)
 */
short WaitForUser(ULONG MySignal)
{
struct Window  *win;
       short   ret=0;

if (IntuitionBase=(struct IntuitionBase *)OpenLibrary("intuition.library",0L))
    {
    if (win=(struct Window *)OpenWindow(&mywin))
        {
        ret=(MySignal==Wait(MySignal | (1L << win->UserPort->mp_SigBit)));
        CloseWindow(win);
        }
    else
        printf("Error: Could not open window\n");
    CloseLibrary((struct Library *)IntuitionBase);
    }
else
    printf("Error: Could not open intution.library\n");
return(ret);
}

VOID main(int argc, char *argv[])
{
struct IOStdReq  *KeyIO;
struct MsgPort   *KeyMP;
struct Interrupt *keyHandler;
struct MyData     MyDataStuff;
       ULONG      MySignal;

if ((MySignal=AllocSignal(-1L))!=-1)
    {
    MyDataStuff.MyTask=FindTask(NULL);
    MyDataStuff.MySignal=1L << MySignal;

    if (KeyMP=CreatePort(NULL,NULL))
        {
        if (keyHandler=AllocMem(sizeof(struct Interrupt),MEMF_PUBLIC|MEMF_CLEAR))
            {
            if (KeyIO=(struct IOStdReq *)CreateExtIO(KeyMP,sizeof(struct IOStdReq)))
                {
                if (!OpenDevice("keyboard.device",NULL,(struct IORequest *)KeyIO,NULL))
                    {
                    keyHandler->is_Code=ResetHandler;
                    keyHandler->is_Data=(APTR)&MyDataStuff;

                    /*
                     * Note that only software interrupt priorities
                     * can be used for the .ln_Pri on the reset
                     * handler...
                     */
                    keyHandler->is_Node.ln_Pri=16;

                    keyHandler->is_Node.ln_Name=NameString;
                    KeyIO->io_Data=(APTR)keyHandler;
                    KeyIO->io_Command=KBD_ADDRESETHANDLER;
                    DoIO((struct IORequest *)KeyIO);

                    if (WaitForUser(MyDataStuff.MySignal))
                        {
                        if (argc) /* Check for CLI */
                            {
                            printf("System going down\n");
                            printf("Cleaning up...\n");
                            /* Show a delay, like cleanup... */
                            Delay(20);
                            printf("*Poof*\n");
                            }
                        /* We are done with our cleanup */

                        KeyIO->io_Data=(APTR)keyHandler;
                        KeyIO->io_Command=KBD_RESETHANDLERDONE;
                        DoIO((struct IORequest *)KeyIO);
                        /*
                         * Note that since the above call
                         * tells the system it is safe to reboot
                         * and will cause the reboot if this
                     * task was the last to say so, the call
                     * never really returns...  The system
                     * just reboots...
                     */
                    }

                KeyIO->io_Data=(APTR)keyHandler;
                KeyIO->io_Command=KBD_REMRESETHANDLER;
                DoIO((struct IORequest *)KeyIO);

                CloseDevice((struct IORequest *)KeyIO);
                }
            else
                printf("Error: Could not open keyboard.device\n");

            DeleteExtIO((struct IORequest *)KeyIO);
            }
        else
            printf("Error: Could not create I/O request\n");

        FreeMem(keyHandler,sizeof(struct Interrupt));
        }
    else
        printf("Error: Could not allocate memory for interrupt\n");

    DeletePort(KeyMP);
    }
else
    printf("Error: Could not create message port\n");

FreeSignal(MySignal);
}
else
    printf("Error: Could not allocate signal\n");
}

****************************************************************************
*       KeyHandler.a
*
* Keyboard reset handler that signals the task in the structure...
*
* See Key_Reset.c for details on how to compile/assemble/link...
*
************************************************************************
* Required includes...
*
        INCDIR  "include:"
        INCLUDE "exec/types.i"
        INCLUDE "exec/io.i"
        INCLUDE "devices/keyboard.i"
*
        xref    _AbsExecBase    ; We get this from outside...
        xref    _LVOSignal      ; We get this from outside...
*
************************************************************************
* Make the entry point external...
*
        xdef    _ResetHandler
*
************************************************************************
*
* This is the input handler
* The is_Data field is passed to you in a1.
*
* This is the structure that is passed in A1 in this example...
*
        STRUCTURE       MyData,0
        APTR            MyTask
        ULONG           MySignal
*
************************************************************************
* The handler gets called here...
*
_ResetHandler:  move.l  MySignal(a1),d0 ; Get signal to send
                move.l  MyTask(a1),a1           ; Get task
*
* Now signal the task...
```

```
*
                move.l  a6,-(sp)        ; Save the stack...
                move.l  _AbsExecBase,a6 ; Get ExecBase
                jsr     _LVOSignal(a6)  ; Send the signal
                move.l  (sp)+,a6        ; Restore A6
*
* Return to let other handlers execute.
*
                rts                     ; return from handler...
*
                END
************************************************************************
```

Reading Keyboard Events

Reading keyboard events is normally not done through direct access to the keyboard device. (Higher level devices such as the input device and console device are available for this. See the chapter "Input Device," for more information on the intimate linkage between the input device and the keyboard device.) This section is provided primarily to show you the component parts of a keyboard input event.

The keyboard matrix figure shown at the beginning of this chapter gives the code value that each key places into the **ie_Code** field of the input event for a key-down event. For a key-up event, a value of hexadecimal 80 is or'ed with the value shown above. Additionally, if either shift key is down, or if the key is one of those in the numeric keypad, the qualifier field of the keyboard input event will be filled in accordingly. In V34 and earlier versions of Kickstart, the keyboard device does not set the numeric qualifier for the keypad keys '(', ')', '/', '*' and '+'.

When you ask to read events from the keyboard, the call will not be satisfied until at least one keyboard event is available to be returned. The **io_Length** field must contain the number of bytes available in **io_Data** to insert events into. Thus, you should use a multiple of the number of bytes in an **InputEvent** (see example below).

> *Type-Ahead Processing.* The keyboard device can queue up several keystrokes without a task requesting a report of keyboard events. However, when the keyboard event buffer has been filled with no task interaction, additional keystrokes will be discarded.

EXAMPLE READ KEYBOARD EVENT PROGRAM

Shown below is an example keyboard.device read-event program:

```
/*
 * Keyboard_Events.c
 *
 * This example does not work very well in a system where
 * input.device is active since input.device also actively calls for
 * keyboard events via this call. For that reason, you will not get all of
 * the keyboard events. Neither will the input device; no one will be happy.
 *
 * Compile with SAS 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <devices/inputevent.h>
```

```
#include <devices/keyboard.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

VOID Display_Event(struct InputEvent *keyEvent)
{
printf("Got key event: KeyCode: %2x  Quailifiers: %4x\n",
               keyEvent->ie_Code,
               keyEvent->ie_Qualifier);
}

VOID main(int argc, char *argv[])
{
struct IOStdReq   *keyRequest;
struct MsgPort    *keyPort;
struct InputEvent *keyEvent;
       SHORT      loop;

if (keyPort=CreatePort(NULL,NULL))
    {
    if (keyRequest=(struct IOStdReq *)CreateExtIO(keyPort,sizeof(struct IOStdReq)))
        {
        if (!OpenDevice("keyboard.device",NULL,(struct IORequest *)keyRequest,NULL))
            {
            if (keyEvent=AllocMem(sizeof(struct InputEvent),MEMF_PUBLIC))
                {
                for (loop=0;loop<4;loop++)
                     {
                     keyRequest->io_Command=KBD_READEVENT;
                     keyRequest->io_Data=(APTR)keyEvent;

                     /*
                      * We want 1 event, so we just set the
                      * length field to the size, in bytes
                      * of the event.  For multiple events,
                      * set this to a multiple of that size.
                      * The keyboard device NEVER fills partial
                      * events...
                      */

                     keyRequest->io_Length=sizeof(struct InputEvent);
                     DoIO((struct IORequest *)keyRequest);

                         /* Check for CLI startup... */
                     if (argc)
                         Display_Event(keyEvent);
                     }

                FreeMem(keyEvent,sizeof(struct InputEvent));
                }
            else
                printf("Error: Could not allocate memory for InputEvent\n");

            CloseDevice((struct IORequest *)keyRequest);
            }
        else
            printf("Error: Could not open keyboard.device\n");

        DeleteExtIO((struct IORequest *)keyRequest);
        }
    else
        printf("Error: Could not create I/O request\n");

    DeletePort(keyPort);
    }
else
    printf("Error: Could not create message port\n");
}
```

Additional Information on the Keyboard Device

Additional programming information on the keyboard device can be found in the include files for the keyboard and input devices and the Autodocs for the keyboard device. All are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Keyboard Device Information	
INCLUDES	devices/keyboard.h devices/keyboard.i devices/inputevent.h devices/inputevent.i
AUTODOCS	keyboard.doc

chapter eight
NARRATOR DEVICE

This chapter describes the narrator device which, together with the translator library, provides all of the Amiga's text-to-speech functions. The narrator device is used to produce high-quality human-like speech in real time.

New Narrator Features for Version 2.0

Feature	Description	Function
NDB_NEWIORB	Flag	Use V37 features
NDB_WORDSYNC	Flag	Synchronize speech/mouth on words
NDB_SYLSYNC	Flag	Synchronize speech/mouth on syllables
F0enthusiasm	**narrator_rb** field	F0 excursion factor
F0perturb	**narrator_rb** field	Amount of F0 perturbation
F1adj	**narrator_rb** field	F1 adjustment in ±5% steps
F2adj	**narrator_rb** field	F2 adjustment in ±5% steps
F3adj	**narrator_rb** field	F3 adjustment in ±5% steps
A1adj	**narrator_rb** field	A1 adjustment in decibels
A2adj	**narrator_rb** field	A2 adjustment in decibels
A3adj	**narrator_rb** field	A3 adjustment in decibels
articulate	**narrator_rb** field	Transition time multiplier
centralize	**narrator_rb** field	Degree of vowel centralization
centphon	**narrator_rb** field	Pointer to central ASCII phon
AVbias	**narrator_rb** field	Amplitude of voicing bias
AFbias	**narrator_rb** field	Amplitude of frication bias
priority	**narrator_rb** field	Priority while speaking

Compatibility Warning: The new features for the 2.0 narrator device are not backwards compatible.

Narrator Device Commands and Functions

Command	Operation
CMD_FLUSH	Purge all active and queued requests for the narrator device.
CMD_READ	Read mouth shapes associated with an active write from the narrator device.
CMD_RESET	Reset the narrator port to its initialized state. All active and queued I/O requests will be aborted. Restarts the device if it has been stopped.
CMD_START	Restart the currently active speech (if any) and resume queued I/O requests.
CMD_STOP	Stop any currently active speech and prevent queued I/O requests from starting.
CMD_WRITE	Write a stream of characters to the narrator device and generate mouth movement data for reads.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the narrator device. If the command is in progress, it is stopped immediately. If it is queued, it is removed from the queue.
BeginIO()	Initiate a command and return immediately (asynchronous request). This is used to minimize the amount of system overhead.
CloseDevice()	Relinquish use of the narrator device. All requests must be complete.
CheckIO()	Return the status of an I/O request.
CloseLibrary()	Relinquish use of a previously opened library.
DoIO()	Initiate a command and wait for completion (synchronous request). Should be used with care because it will not return control if the request does not complete.
OpenDevice()	Obtain use of the narrator device.
OpenLibrary()	Obtain use of a library.
SendIO()	Initiate a command and return immediately (asynchronous request).
WaitIO()	Wait for the completion of an asynchronous request. When the request is complete the message will be removed from reply port.

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure of type **narrator_rb**. This structure will be used to communicate commands to the narrator device.
CreatePort()	Create a signal message port for reply messages from the narrator device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete an extended I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The narrator device operates like all other Amiga devices. To use the narrator device, you must first open it. This initializes certain global areas, opens the audio device, allocates audio channels, and performs other housekeeping functions. Once open, the device is ready to receive I/O commands (most typically CMD_WRITE and CMD_READ). Finally, when finished, the user should close the device. This will free some buffers and allow the entire device to be expunged should the system require memory. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The narrator device uses two extended I/O request structures: **narrator_rb** for write commands (to produce speech output) and **mouth_rb** for read commands (to receive mouth shape changes and word/syllable synchronization events). Both I/O request structures have been expanded (in a backwards compatible fashion) for the V37 narrator device with several new fields defined.

```
struct narrator_rb
{
    struct IOStdReq  message;    /* Standard IORequest Block     */
    UWORD   rate;                /* Speaking rate (words/minute) */
    UWORD   pitch;               /* Baseline pitch in Hertz      */
    UWORD   mode;                /* Pitch mode                   */
    UWORD   sex;                 /* Sex of voice                 */
    UBYTE   *ch_masks;           /* Pointer to audio allocation maps  */
    UWORD   nm_masks;            /* Number of audio allocation maps   */
    UWORD   volume;              /* Volume. 0 (off) thru 64      */
    UWORD   sampfreq;            /* Audio sampling frequency     */
    UBYTE   mouths;              /* If non-zero, generate mouths */
    UBYTE   chanmask;            /* Which ch mask used (internal - do not modify)*/
    UBYTE   numchan;             /* Num ch masks used (internal- do not modify) */
    UBYTE   flags;               /* New feature flags            */
    UBYTE   F0enthusiasm;        /* F0 excursion factor          */
    UBYTE   F0perturb;           /* Amount of F0 perturbation    */
    BYTE    F1adj;               /* F1 adjustment in +- 5% steps   */
    BYTE    F2adj;               /* F2 adjustment in +- 5% steps   */
    BYTE    F3adj;               /* F3 adjustment in +- 5% steps   */
    BYTE    A1adj;               /* A1 adjustment in decibels    */
    BYTE    A2adj;               /* A2 adjustment in decibels    */
    BYTE    A3adj;               /* A3 adjustment in decibels    */
    UBYTE   articulate;          /* Transition time multiplier   */
    UBYTE   centralize;          /* Degree of vowel centralization */
    char    *centphon;           /* Pointer to central ASCII phon  */
    BYTE    AVbias;              /* Amplitude of voicing bias    */
    BYTE    AFbias;              /* Amplitude of frication bias  */
    BYTE    priority;            /* Priority while speaking      */
    BYTE    pad1;                /* For alignment                */
};

struct mouth_rb
{
    struct  narrator_rb voice;   /* Speech IORequest Block       */
    UBYTE   width;               /* Mouth width (returned value) */
    UBYTE   height;              /* Mouth height (returned value)*/
    UBYTE   shape;               /* Internal use, do not modify  */
    UBYTE   sync;                /* Returned sync events         */
};
```

Details on the meaning of the various fields of the two I/O request blocks can be found in the "Writing to the Narrator Device" and "Reading from the Narrator Device" sections later in this chapter. See the include file *devices/narrator.h* for the complete structure definitions.

THE AMIGA SPEECH SYSTEM

The speech system on the Amiga is divided into two subsystems:

- The translator library, consisting of a single function: **Translate()**, which converts an English string into its phonetic representation, and
- The narrator device, which uses the phonetic representation (generated either manually or by the translator library) as input to generate human-like speech and play it out via the audio device.

The two subsystems can be used either together or individually. Generally, hand coding phonetic text will produce better quality speech than using the translator library, but this requires the programmer to "hard code" the phonetic text in the program or otherwise restrict the input to phonetic text only. If the program must handle arbitrary English input, the translator library should be used.

Below is an example of how you would use the translator library to translate a string for the narrator device.

```
#define BUFLEN 500

APTR EnglStr;                  /* pointer to sample input string */
LONG EnglLen;                  /* input length */
UBYTE PhonBuffer[BUFLEN];      /* place to put the translation */
LONG rtnCode;                  /* return code from function */

struct narrator_rb *VoiceIO;   /* speaking I/O request block */
struct mouth_rb *MouthIO;      /* mouth movement I/O request block */

EnglStr = "This is Amiga speaking.";    /* a test string */
EnglLen = strlen(EnglStr);
rtnCode = Translate(EnglStr, EnglLen, (APTR)&PhonBuffer[0], BUFLEN);

voice_io->message.io_Command = CMD_WRITE;
voice_io->message.io_Offset  = 0;
voice_io->message.io_Data    = PhonBuffer;
voice_io->message.io_Length  = strlen(PhonBuffer);
DoIO((struct IORequest *)VoiceIO)
```

This chapter discusses only the narrator device; refer to the "Translator Library" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information on the translator library.

While the narrator device on the Amiga supports all of the major device commands (see the Narrator Device Commands and Functions section), two of these commands do most of the work in the device. They are:

- CMD_WRITE—This command is used to send a phonetic string to the device to be spoken. The **narrator_rb** I/O request block also contains several parameters which can be set to control various aspects of the speech, such as pitch, speaking rate, male/female voice, and so on. Some of the options are rather arcane. See the "Writing to the Narrator Device" section for a complete list of options and their descriptions.
- CMD_READ—The narrator device can be told to generate various synchronization events which the user can query. These events are: mouth shape changes, word sync, and/or syllable sync. The events can be generated singly or in any combination, as requested by the user. Word and syllable synchronization events are new to system 2.0 and later (V37 and later of the narrator device). See the "Reading from the Narrator Device" section for more details.

OPENING THE NARRATOR DEVICE

Three primary steps are required to open the narrator device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an extended I/O request structure of type **narrator_rb**. The **narrator_rb** structure is created by the **CreateExtIO()** function.
- Open the narrator device. Call **OpenDevice()** passing the I/O request.

```
struct MsgPort *VoiceMP;
struct narrator_rb *VoiceIO;

if (VoiceMP = CreatePort("speech_write",0))
    if (VoiceIO = (struct narrator_rb *)
                      CreateExtIO(VoiceMP,sizeof(struct narrator_rb));
        if (OpenDevice("narrator.device", 0, VoiceIO, 0))
                printf("narrator.device did not open\n");
```

When the narrator device is first opened, it initializes certain fields in the user's **narrator_rb** I/O request structure. In order to maintain backwards compatibility with older versions of the narrator device, a mechanism was needed for the device to ascertain whether it was being opened with a V37 or pre-V37 style I/O request structure. The pad field in the pre-V37 **narrator_rb** I/O request structure (which no one should have ever touched!) has been replaced by the flags field in the V37 **narrator_rb** structure, and is our path to upward compatibility. The device checks to see if a bit is set in this flags field. *This bit must be set before opening the device if V37 or later features of the narrator device are to be used.* There are two defined constants in the include file, NDB_NEWIORB and NDF_NEWIORB. NDB_NEWIORB specifies the bit which must be set in the flags field, NDF_NEWIORB is the field definition of the bit (1 << NDB_NEWIORB).

Once the device is opened, the **mouth_rb** (read) I/O request structure can be set up. Each CMD_READ request must be matched with an associated CMD_WRITE request. This is necessary for the device to match the various sync events with a particular utterance. The read I/O request structure is easily set up as follows:

- Create a read message port using the **CreatePort()** function.
- Allocate memory for the **mouth_rb** extended I/O request structure using **AllocMem()**.
- Copy the **narrator_rb** I/O request structure used to open the device into the voice field of the **mouth_rb** I/O request structure. This will set the fields necessary for the device to make the correct correspondence between read and write requests.
- Copy the pointer to the read message port returned from **CreatePort()** into the **voice.message.io_Message.mn_ReplyPort** field of the **mouth_rb** structure.

The following code fragment, in conjunction with the **OpenDevice()** code fragment above, shows how to set up the **mouth_rb** structure:

```
struct  MsgPort   *MouthMP;
struct  mouth_rb  *MouthIO;

if (MouthMP = CreatePort("narrator_read", 0))
    if (!(MouthIO = (struct mouth_rb *)
                    AllocMem(sizeof(struct mouth_rb),MEMF_PUBLIC|MEMF_CLEAR)))
        {
        MouthIO->voice = *VoiceIO;       /* Copy I/O request used in OpenDevice */
        MouthIO->voice.message.io_Message.mn_ReplyPort = MouthMP; /* Set port */
        }
    else
        printf("AllocMem failed\n");
else
    printf("CreatePort failed\n");
```

CLOSING THE NARRATOR DEVICE

Each **OpenDevice()** must be eventually matched by a call to **CloseDevice()**. This is necessary to allow the system to expunge the device in low memory conditions. As long as any task has the device open, or has forgotten to close it before terminating, the narrator device will not be expunged.

All I/O requests must have completed before the task can close the device. If any requests are still pending, the user must abort them before closing the device.

```
if (!(CheckIO(VoiceIO)
    {
    AbortIO(VoiceIO);  /* Abort queued or in progress request */
    }
WaitIO((struct IORequest *)VoiceIO);       /* Wait for abort to do its job */
CloseDevice(VoiceIO);                      /* Close the device */
```

Writing to the Narrator Device

You write to the narrator device by passing a **narrator_rb** I/O request to the device with CMD_WRITE set in **io_Command**, the number of bytes to be written set in **io_Length** and the address of the write buffer set in **io_Data**.

```
VoiceIO->message.io_Command = CMD_WRITE;
VoiceIO->message.io_Offset  = 0;
VoiceIO->message.io_Data    = PhonBuffer;
VoiceIO->message.io_Length  = strlen(PhonBuffer);
DoIO((struct IORequest *)VoiceIO);
```

You can control several characteristics of the speech, as indicated in the **narrator_rb** struct shown in the "Device Interface" section.

Generally, the narrator device attempts to speak in a non-regional dialect of American English. With pre-V37 versions of the device, the user could change only a few of the more basic aspects of the speaking voice such as pitch, male/female, speaking rate, etc. With the V37 and later versions of the narrator device, the user can now change many more aspects of the speaking voice. In addition, in the pre-V37 device, only mouth shape changes could be queried by the user. With the V37 device, the user can also receive start of word and start of syllable synchronization events. These events can be generated independently, giving the user much greater flexibility in synchronizing voice to animation or other effects.

The following describes the fields of the **narrator_rb** structure:

message.io_Data
Points to a NULL-terminated ASCII phonetic input string. For backwards compatibility issues, the string may also be terminated with a '#' symbol. See the "How to Write Phonetically for Narrator" section of this chapter for details.

message.io_Length
Length of the input string. The narrator device will parse the input string until either a NULL or a '#' is encountered, or until io_Length characters have been processed.

rate
The speaking rate in words/minute. Range is from 40 to 400 wpm.

pitch
The baseline pitch of the speaking voice. Range is 65 to 320 Hertz.

mode
The F0 (pitch) mode. ROBOTICF0 produces a monotone pitch, NATURALF0 produces a normal pitch contour, and MANUALF0 (new for V37 and later) gives the user more explicit control over the pitch contour by creative use of accent numbers. In MANUALF0 mode, a given accent number will have the same effect on the pitch regardless of its position in the sentence and its relation to other accented syllables. In NATURALF0 mode, accent numbers have a reduced effect towards the end of sentences (especially long ones). In addition, the proximity of other accented syllables, the number of syllables in the word, and the number of phrases and words in the sentence all affect the pitch contour. In MANUALF0 mode these things are ignored and it's up to the user to do the controlling. This has the advantage of being able to have the pitch be more expressive. The F0enthusiasm field will scale the effect.

sex
Controls the sex of the speaking voice (MALE or FEMALE). In actuality, only the formant targets are changed. The user must still change the pitch and speaking rate of the voice to get the correct sounding sex. See the include files for default pitch and rate settings.

ch_masks
Pointer to a set of audio allocation maps. See the "Audio Device" chapter for details.

nm_masks
Number of audio allocation maps. See the "Audio Device" chapter for details.

volume
Sets the volume of the speaking voice. Range 0 - 64.

sampfreq
The synthesizer is "tuned" to a sampling frequency of 22,200 Hz. Changing sampfreq affects pitch and formant tunings and can be used to create unusual vocal effects. For V37 and later, it is recommended that F1, F2, and F3adj be used instead to achieve this effect.

mouths
If set to a non-zero value will direct the narrator device to generate mouth shape changes and send this data to the user in response to read requests. See the "Reading from the Narrator Device" section for more details.

chanmask
Used internally by the narrator device. The user should not modify this field.

numchan
Used internally by the narrator device. The user should not modify this field.

flags (V37)
Used to specify V37 features of the device. Possible bit settings are:
NDB_NEWIORB - I/O request block uses V37 features.
NDB_WORDSYNC - Device should generate start of word sync events.
NDB_SYLSYNC - Device should generate start of syllable sync events.
These bit definitions and their corresponding field definitions (NDF_NEWIORB, NDF_WORDSYNC, and NDF_SYLSYNC) can be found in the include files.

F0enthusiasm (V37)
The value of this field controls the scaling of pitch (F0) excursions used on accented syllables and has the effect of making the narrator device sound more or less "enthusiastic" about what it is saying. It is calibrated in 1/32s with unity (32) being the default value. Higher values cause more F0 variation, lesser values cause less. This feature is most useful in manual F0 mode.

F0perturb (V37)
Non-zero values in this field cause varying amounts of random low-frequency modulation of the pitch (F0). In other words, the pitch shakes in much the same way as an elderly person's voice does. Range is 0 to 255.

F1adj, F2adj, F3adj (V37)
Changes the tuning of the formant frequencies. A formant is a major vocal tract resonance, and the frequencies of these formants move continuously as we speak. Traditionally, they have been given the abbreviations of F1, F2, F3... with F1 being the one lowest in frequency. Moving these formants away from their normal positions causes drastic changes in the sound of the voice and is a very powerful tool in the creation of character voices. This adjustment is in ±5% steps. Positive values raise the formant frequencies and vice versa. The default is zero. Use these adjustments instead of changing sampfreq.

A1adj, A2adj, A3adj (V37)
In a parallel formant synthesizer, the amplitudes of the formants need to be specified along with their frequencies. These fields bias the amplitudes computed by the narrator device. This is useful for creating different tonal balances (bass or treble), and listening to formants in isolation for educational purposes. The adjustments are calibrated directly in ±1db (decibel) steps. Using negative values will cause no problems; use of positive numbers can cause clipping. If you want to raise an amplitude, try cutting the others the same relative amount, then bring them all up equally until clipping is heard, then back them off. This should produce an optimum setting. This field has a +31 to -32 db range and the value -32db is equivalent to -infinity, shutting that formant off completely.

articulate (V37)
According to the popular theories of speech production, we move our articulators (jaw, tongue, lips, etc.) smoothly from one "target" position to the next. These articulatory targets correspond to acoustic targets specified by the narrator device for each phoneme. The device calculates the time it should take to get from one target to the next and this field allows you to intervene in that process. Values larger than the default will cause the transitions to be proportionately longer and vice versa. This field is calibrated in percent with 100 being the default. For example, a value of 50 will cause the transitions to take half the normal time, with the result being "sharper", more deliberate sounding speech (not necessarily more natural). A value of

200 will cause the transitions to be twice as long, slurring the speech. Zero is a special value in the narrator device will take special measures to create *no* transitions at all and each phoneme will simply be abutted to the next.

centralize (V37)

This field together with centphon can be used to create regional accent effects by modifying vowel sounds. centralize specifies the degree (in percent) to which vowel targets are "pulled" towards the targets of the vowel specified by centphon. The default value of 0% indicates that each vowel in the utterance retains its own target values. The maximum value of 100% indicates that each vowel's targets are replaced by the targets of the specified vowel. Intermediate values control the degree of interpolation between the utterance vowel's targets and the targets of the vowel specified by centphon.

centphon (V37)

Pointer to an ASCII string specifying the vowel whose targets are used in the interpolation specified by centralize. The vowels which can be specified are: IY, IH, EH, AE, AA, AH, AO, OW, UH, ER, UW. Specifying other than these will result in an error code being returned.

AVbias, AFbias (V37)

Controls the relative amplitudes of the voiced and unvoiced speech sounds. Voiced sounds are those made with the vocal cords vibrating, such as vowels and some consonants like y, r, w, and m. Unvoiced sounds are made without the vocal cords vibrating and use the sound of turbulent air, such as s, t, sh, and f. Some sounds are combinations of both such as z and v. AVbias and AFbias change the default amplitude of the voiced and unvoiced components of the sounds respectively. (AV stands for Amplitude of Voicing and AF stands for Amplitude of Frication). These fields are calibrated in ±1db steps and have the same range as the other amplitude biases, namely +31 to -32 db. Again, positive values may cause clipping. Negative values are the most useful.

priority (V37)

Task priority while speaking. When the narrator device begins to synthesize a sentence, the task priority remains unchanged while it is calculating acoustic parameters. However, when speech begins at the end of this process, the priority is bumped to 100 (the default value). If you wish, you may change this to anything you want. Higher values will tend to lock out most anything while speech is going on, and lower values may cause audible breaks in the speech output. The following example shows how to issue a write request to the narrator device. The first write is done with the default parameter settings. The second write is done after modifying the first and third formant loudness and using the centralization feature.

The following example shows how to issue a write request to the narrator device. The first write is done with the default parameter settings. The second write is done after modifying the first and third formant loudness and using the centralization feature.

```
/*
 * Speak_Narrator.c
 *
 * This example program sends a string of phonetic text to the narrator
 * device twice, changing some of the characteristics the second time.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Requires Kickstart V37 or greater.
 */

#include <exec/types.h>
```

```
#include <exec/exec.h>
#include <dos/dos.h>
#include <devices/narrator.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>

#include <string.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void main(void)
{
struct  MsgPort     *VoiceMP;
struct  narrator_rb *VoiceIO;
UBYTE   *PhoneticText    = "DHIHS IHZ AHMIY5GAH SPIY5KIHNX.";
BYTE    audio_chan[4]    = {3, 5, 10, 12};

    /* Create the message port */
if (VoiceMP=CreateMsgPort())
    {
        /* Create the I/O request */
    if (VoiceIO = CreateIORequest(VoiceMP,sizeof(struct narrator_rb)))
        {
        /*   Set the NEWIORB bit in the flags field to use the new fields */
        VoiceIO->flags = NDF_NEWIORB;

            /* Open the narrator device */
        if (OpenDevice("narrator.device",0,(struct IORequest *)VoiceIO,0L))

            /* Inform user that it could not be opened */
            printf("Error: narrator.device did not open\n");

        else
            {
             /* Speak the string using the default parameters */
             VoiceIO->ch_masks = &audio_chan[0];
             VoiceIO->nm_masks = sizeof(audio_chan);
             VoiceIO->message.io_Command = CMD_WRITE;
             VoiceIO->message.io_Data = PhoneticText;
             VoiceIO->message.io_Length = strlen(PhoneticText);
             DoIO(VoiceIO);

             /* Now change some of the characteristics:
              *    Raise the first formant, lower the third formant,
              *    and move 50% of the way towards AO.
              * and speak it again.
              */

            VoiceIO->A1adj = -32;                        /* Shut off first formant  */
            VoiceIO->A3adj =  11;                        /* Raise the third formant */
            VoiceIO->centralize = 50;            /* Move 50% of the way     */
            VoiceIO->centphon = "AO";            /* towards AO              */
            DoIO(VoiceIO);

            /* Close the narrator device */
            CloseDevice((struct IORequest *)VoiceIO);
            }
        /* Delete the IORequest */
        DeleteIORequest(VoiceIO);
        }
    else
        /* Inform user that the I/O request could be created */
        printf("Error: Could not create I/O request\n");

    /* Delete the message port */
    DeleteMsgPort(VoiceMP);
    }
else
    /* Inform user that the message port could not be created */
    printf("Error: Could not create message port\n");
}
```

Reading from the Narrator Device

All read requests to the narrator device must be matched to an associated write request. This is done by copying the **narrator_rb** structure used in the **OpenDevice()** call into the **voice** field of the **mouth_rb** I/O request structure. You must do this *after* the call to **OpenDevice()**. Matching the read and write requests allows the narrator device to coordinate I/O requests across multiple uses of the device.

In pre-V37 versions of the narrator device, only mouth shape changes can be queried from the device. This is done by setting the **mouths** field of the **narrator_rb** I/O request structure (the write request) to a non-zero value. The write request is then sent asynchronously to the device and while it is in progress, synchronous read requests are sent to the device using the **mouth_rb** I/O request structure. When the mouth shape has changed, the device will return the read request to the user with bit 0 set in the **sync** field of the **mouth_rb**. The fields **width** and **height** of the **mouth_rb** structure will contain byte values which are proportional to the actual width and height of the mouth for the phoneme currently being spoken. Read requests sent to the narrator device are not returned to the user until one of two things happen: either the mouth shape has changed (this prevents the user from having to constantly redraw the same mouth shape), or the speech has completed. The user can check **io_Error** to determine if the mouth shape has changed (a return code of 0) or if the speech has completed (return code of ND_NoWrite).

In addition to returning mouth shapes, reads to the V37 narrator device can also perform two new functions: word and syllable sync. To generate word and/or syllable sync events, the user must specify several bits in the **flags** field of the write request (**narrator_rb** structure). The bits are NDB_WORDSYNC and NDB_SYLSYNC, for start of word and start of syllable synchronization events, respectively, and, of course, NDB_NEWIORB, to indicate that the V37 I/O request is required.

NDB_WORDSYNC and NDB_SYLSYNC tell the device to expect read requests and to generate the appropriate event(s). As with mouth shape change events, the write request is sent asynchronously to the device and, while it is in progress, synchronous read requests are sent to the device. The **sync** field of the **mouth_rb** structure will contain flags indicating which events (mouth shape changes, word sync, and/or syllable sync) have occurred.

The returned **sync** field flags are:

bit 0 (0x01) $\Longrightarrow$ mouth shape change event
bit 1 (0x02) $\Longrightarrow$ start-of-word synchronization event
bit 2 (0x04) $\Longrightarrow$ start-of-syllable synchronization event

and 1 or more flags may be set for any particular read.

As with mouth shape changes, read requests will not return until the requested event(s) have occurred, and the user must test the **io_Error** field of the **mouth_rb** structure to tell when the speech has completed (an error return of ND_NoWrite).

Several read events can be compressed into a single event. This can occur in two ways: first when two dissimilar events occur between two successive read requests. For example, a single read may return both a mouth change and a syllable sync event. This should not present a problem if the user checks for all events. The second is when multiple events of the same type occur between successive read requests. This is of no great concern in dealing with mouth shape changes because, presumably, mouth events are used to drive animation, and the animation procedure will simply draw the current mouth shape.

Watch Those Sync Events. When word or syllable sync is desired, the narrator device may compress multiple sync events into a single sync event. Missing a word or syllable sync may cause word highlighting (for example) to lose sync with the speech output. A future version of the device will include an extension to the **mouth_rb** I/O request structure which will contain word and syllable counts and, possibly, other synchronization methods.

The following code fragment shows the basics of how to perform reads from the narrator device. For a more complete example, see the sample program at the end of this chapter. For this fragment, take the code of the previous write example as a starting point. Then the following code would need to be added:

```
struct  mouth_rb  *MouthIO;      /* Pointer to read IORequest block */
struct  MsgPort   *MouthMP;      /* Pointer to read message port    */

/*
 *  (1) Create a message port for the read request.
 */
if (!(MouthMP = CreatePort("narrator_read", 0L)))
    BellyUp("Read CreatePort failed");

/*
 *  (2) Create an extended IORequest of type mouth_rb.
 */
if (!(MouthIO = (struct mouth_rb *)CreateExtIO(MouthMP, sizeof(struct mouth_rb))))
    BellyUp("Read CreateExtIO failed");

/*
 *  (3) Set up the read IORequest. Must be done after the call to OpenDevice().
 *      We assume that the write IORequest and the OpenDevice have been done
 */
MouthIO->voice  =  *SpeakIO;
MouthIO->voice.message.io_Message.mn_ReplyPort = ReadMsgPort;
MouthIO->voice.message.io_Command = CMD_READ;

/*
 *  (4) Set the flags field of the narrator_rb write request to return the desired
 *      sync events.  If mouth shape changes are required, then the mouths field
 *      of the IORequest should be set to a non-zero value.
 */

SpeakIO->mouths = 1;              /* Generate mouth shape changes */
SpeakIO->flags = NDF_NEWIORB |    /* Indicates V37 style IORequest */
                 NDF_WORDSYNC |   /* Request start-of-word sync events   */
                 NDF_SYLSYNC;     /* Request start-of-syllable sync events */

/*
 *  (5) Issue asynchronous write request.  The driver initiates the write request
 *      and returns immediately.
 */

SendIO(SpeakIO);

/*
 *  (6) Issue synchronous read requests.  For each request we check the sync field
 *      to see which events have occurred.  Since any combination of events can
 *      be returned in a single read, we must check all possibilities.  We
 *      continue looping until the read request returns an error of ND_NoWrite,
 *      which indicates that the write request has completed.
 */

for   (DoIO(MouthIO);MouthIO->voice.message.io_Error != ND_NoWrite;DoIO(MouthIO))
      {
      if (MouthIO->sync & 0x01)  DoMouthShape();
      if (MouthIO->sync & 0x02)  DoWordSync();
      if (MouthIO->sync & 0x04)  DoSyllableSync();
      }

/*
 *  (7) Finally, we must perform a WaitIO() on the original write request.
 */

WaitIO(SpeakIO);
```

How to Write Phonetically for Narrator

This section describes in detail the procedure used to specify phonetic strings to the *narrator* speech synthesizer. No previous experience with phonetics is required. The only thing you may need is a good pronunciation dictionary for those times when you doubt your own ears. You do not have to learn a foreign language or computer language. You are just going to learn how to write down the English that comes out of your own mouth. In writing phonetically you do not have to know how a word is spelled, just how it is said.

Table of Phonemes

Vowels

Phoneme	Example	Phoneme	Example
IY	beet, eat	IH	bit, in
EH	bet, end	AE	bat, ad
AA	bottle, on	AH	but, up
AO	ball, awl	UH	book, soot
ER	bird, early	OH	border
AX*	about, calibrate	IX*	solid, infinite

* AX and IX should never be used in stressed syllables.

Diphthongs

Phoneme	Example	Phoneme	Example
EY	bay,aid	AY	bide,I
OY	boy,oil	AW	bound,owl
OW	boat,own	UW	brew,boolean

Consonants

Phoneme	Example	Phoneme	Example
R	red	L	long
W	wag	Y	yellow,comp(Y)uter
M	men	N	no
NX	sing	SH	shy
S	soon	TH	thin
F	fed	ZH	pleasure
Z	has,zoo	DH	then
V	very	WH	when
CH	check	J	judge
/H	hole	/C	loch
B	but	P	put
D	dog	T	toy
K	keg,copy	G	guest

Special Symbols

Phoneme	Example	Explanation
DX	pity	tongue flap
Q	kitt(Q)en	glottal stop
QX		silent vowel

Contractions (see text)

UL = AXL
IL = IXL
UM = AXM
IM = IXM
UN = AXN
IN = IXN

Digits and Punctuation

Digits 1-9	Syllabic stress, ranging from secondary through emphatic
.	Period - sentence final character.
?	Question mark - sentence final character
-	Dash - phrase delimiter
,	Comma - clause delimiter
()	Parentheses - noun phrase delimiters (see text)

The narrator device works on utterances at the sentence level. Even if you want to say only one word, it will treat it as a complete sentence. Therefore, narrator wants one of two punctuation marks to appear at the end of every sentence - a period or a question mark. The period is used for almost all utterances and will cause a final fall in pitch to occur at the end of a sentence. The question mark is used at the end of yes/no questions only, and results in a final rise in pitch.

For example, the question, *Do you enjoy using your Amiga?* would take a question mark at the end, while the question, *What is your favorite color?* should be followed (in the phonetic transcription) with a period. If no punctuation appears at the end of a string, narrator will append a dash to it, which will result in a short pause. Narrator recognizes other punctuation marks as well, but these are left for later discussion.

PHONETIC SPELLING

Utterances are usually written phonetically using an alphabet of symbols known as IPA (International Phonetic Alphabet). This alphabet is found at the front of most good dictionaries. The symbols can be hard to learn and were not readily available on computer keyboards, so the Advanced Research Projects Agency (ARPA) came up with the ARPABET, a way of representing each symbol using one or two upper case letters. Narrator uses an expanded version of the ARPABET to specify phonetic sounds.

A phonetic sound, or **phoneme**, is a basic speech sound, a speech atom. Working backwards: sentences can be broken into words, words into syllables, and syllables into phonemes. The word *cat* has three letters and (coincidentally) three phonemes. Looking at the table of phonemes we find the three sounds that make up the word *cat*. They are the phonemes **K, AE,** and **T**, written as **KAET**. The word *cent* translates as **SEHNT**. Notice that both words begin with the letter *c*, but because they are pronounced differently they have different phonetic spellings. These examples introduce a very important concept of phonetic spelling: spell it like it sounds, not like it looks.

Choosing the Right Vowel

Phonemes, like letters, are divided into two categories: vowels and consonants. Loosely defined, a vowel is a continuous sound made with the vocal cords vibrating and air exiting the mouth (as opposed to the nose). A consonant is any other sound, such as those made by rushing air (like S or TH), or by interruptions in the air flow by the lips or tongue (B or T). All vowels use a two letter ASCII phonetic code while consonants use a one or two letter code.

In English we write with only five vowels: a, e, i, o, and u. It would be easy if we only said five vowels. However, we say more than 15 vowels. Narrator provides for most of them. Choose the proper vowel by listening: Say the word aloud, perhaps extending the vowel sound you want to hear and then compare the sound you are making to the sounds made by the vowels in the examples on the phoneme list. For example, the *a* in *apple* sounds the same as the *a* in *cat*, not like the *a* in *Amiga, talk,* or *made*. Notice also that some of the example words in the list do not even use any of the same letters contained in the phoneme code; for example **AA** as in *bottle*.

Vowels are divided into two groups: those that maintain the same sound throughout their durations and those that change their sound. The ones that change are called **diphthongs**. Some of us were taught the terms long and short to describe vowel sounds. Diphthongs fall into the long category, but these two terms are inadequate to fully differentiate between vowels and should be avoided. The diphthongs are the last six vowels listed in the table. Say the word *made* out loud very slowly. Notice how the *a* starts out like the *e* in *bet* but ends up like the *e* in *beet*. The *a*, therefore, is a diphthong in this word and we would use **EY** to represent it. Some speech synthesis systems require you to specify the changing sounds in diphthongs as separate elements, but narrator takes care of the assembly of diphthongal sounds for you.

Choosing the Right Consonant

Consonants are divided into many categories by phoneticians, but we need not concern ourselves with most of them. Picking the correct consonant is very easy if you pay attention to just two categories: voiced and unvoiced. A voiced consonant is made with the vocal cords vibrating, and an unvoiced one is made when the vocal cords are silent. Sometimes English uses the same letter combinations to represent both. Compare the *th* in *thin* with the *th* in *then*. Notice that the first is made with air rushing between the tongue and upper teeth. In the second, the vocal cords are vibrating also. The voiced *th* phoneme is **DH** and the unvoiced one is **TH.** Therefore, *thin* is phonetically spelled as **THIHN** while the word *then* is spelled **DHEHN**.

A sound that is particularly subject to mistakes is voiced and unvoiced *s*, phonemes **Z** and **S**, respectively. Clearly the word *bats* ends with an S and the word has ends with a **Z.** But, how do you spell *close*? If you say "What time do you close?", you spell it with a **Z**, and if you are saying "I love to be close to you." you use an S."

Another sound that causes some confusion is the *r* sound. There are two different r-like phonemes in the Narrator alphabet: **R** under the consonants and **ER** under the vowels. Use **ER** if the *r* sound is the vowel sound in the syllable like in *bird, absurd,* and *flirt*. Use the **R** if the *r* sound precedes or follows another vowel sound in that syllable as in *car, write,* and *craft*.

Contractions and Special Symbols

There are several phoneme combinations that appear very often in English words. Some of these are caused by our laziness in pronunciation. Take the word *connector* for example. The *o* in the first syllable is almost swallowed out of existence. You would not use the **AA** phoneme; you would use the **AX** phoneme instead. It is because of this relaxation of vowels that we find ourselves using **AX** and **IX** very often. Since this relaxation frequently occurs before *l*, *m*, and *n*, narrator has a shortcut for typing these combinations. Instead of *personal* being spelled **PERSIXNAXL**, we can spell it **PERSINUL**, making it a little more readable. *Anomaly* goes from **AXNAAMAXLIY** to **UNAAMULIY**, and **KAAMBIXNEYSHIXN** becomes **KAAMBINEYSHIN** for combination. It may be hard to decide whether to use the **AX** or **IX** brand of relaxed vowel. The only way to find out is to use both and see which sounds best.

Other special symbols are used internally by narrator. Sometimes they are inserted into or substituted for part of your input sentence. You can type them in directly if you wish. The most useful is probably the **Q** or glottal stop, an interruption of air flow in the glottis. The word *Atlantic* has one between the *t* and the *l*. Narrator knows there should be a glottal stop there and saves you the trouble of typing it. But narrator is only close to perfect, so sometimes a word or word pair might slip by that would have sounded better with a **Q** stuck in someplace.

STRESS AND INTONATION

It is not enough to tell narrator what you want said. For the best results you must also tell narrator how you want it said. In this way you can alter a sentence's meaning, stress important words, and specify the proper accents in polysyllabic words. These things improve the naturalness and thus the intelligibility of the spoken output.

Stress and intonation are specified by the single digits 1-9 following a vowel phoneme code. Stress and intonation are two different things, but are specified by a single number.

Stress is, among other things, the elongation of a syllable. A syllable is either stressed or not, so the presence of a number after the vowel in a syllable indicates stress on that syllable. The value of the number indicates the intonation. These numbers are referred to here as stress marks but keep in mind that they also affect intonation.

Intonation here means the pitch pattern or contour of an utterance. The higher the stress mark, the higher the potential for an accent in pitch. A sentence's basic contour is comprised of a quickly rising pitch gesture up to the first stressed syllable in the sentence, followed by a slowly declining tone throughout the sentence, and finally, a quick fall to a low pitch on the last syllable. The presence of additional stressed syllables causes the pitch to break its slow, declining pattern with rises and falls around each stressed syllable. Narrator uses a very sophisticated procedure to generate natural pitch contours based on how you mark the stressed syllables.

How and Where to Put the Stress Marks

The stress marks go immediately to the right of vowel phoneme codes. The word *cat* has its stress marked after the **AE**, e.g., **KAE5T**. You generally have no choice about the location of a number; there is definitely a right and wrong location. A number should either go after a vowel or it should not. Narrator will not flag an error if you forget to put a stress mark in or if you place it on the wrong vowel. It will only tell you if a stress mark has been put after a non-vowel, i.e., consonant or punctuation.

The rules for placing stress marks are as follows:

- Always place a stress mark in a content word. A content word is one that contains some meaning. Nouns, verbs, and adjectives are all content words, they tell the listener what you are talking about. Words like *but*, *if*, and **the** are not content words. They do not convey any real world meaning, but are required to make the sentence function, so they are given the name function words.
- Always place a stress mark on the accented syllable(s) of polysyllabic words, whether they are content or function words. A polysyllabic word is any word of more than one syllable. *Commodore* has its stress (often called accent) on the first syllable and would be spelled **KAA5MAXDOHR**, while *computer* is stressed on the second syllable: **KUMPYUW5TER**.

If you are in doubt about which syllable gets the stress, look up the word in a dictionary and you will find an accent mark over the stressed syllable. If more than one syllable in a word receives stress, they usually are not of equal value. These are referred to as primary and secondary stresses. The word *understand* has its first and last syllables stressed, with the syllable *stand* getting the primary stress and the syllable *un* getting the secondary stress. This produces the phonetic representation **AH1NDERSTAE4ND**. Syllables with secondary stress should be marked with a value of only 1 or 2.

Compound words (words with more than one root) such as *baseball*, *software*, and *lunchwagon* can be written as one word, but should be thought of as separate words when marking stress. Thus, lunchwagon would be spelled **LAH5NCHWAE2GIN**. Notice that the *lunch* got a higher stress mark than the *wagon*. This is common in compound words, the first word usually receives the primary stress.

Which Stress Value Do I Use?

If you get the spelling and stress mark positions correct, you are 95 percent of the way to a good sounding sentence. The next thing to do is decide on the stress mark values. They can be roughly related to parts of speech, and you can use the table shown below as a guide to assigning values.

Recommended Stress Values

Part of Speech	Stress Value
Exclamations	9
Adverbs	7
Quantifiers	7
Nouns	5
Adjectives	5
Verbs	4
Pronouns	3
Secondary stress	1 or 2
Everything else	None

The above values merely suggest a range. If you want attention directed to a certain word, raise its value. If you want to downplay a word, lower it. Sometimes even a function word can be the focus of a sentence. It is quite conceivable that the word *to* in the sentence *Please deliver this to Mr. Smith.* could receive a stress mark of 9. This would add focus to the word, indicating that the item should be delivered to Mr. Smith in person.

PUNCTUATION

In addition to the period or question mark that is required at the end of a sentence, Narrator also recognizes dashes, commas, and parentheses.

The comma goes where you would normally put a comma in an English sentence. It causes narrator to pause with a slightly rising pitch, indicating that there is more to come. The use of additional commas—that is, more than would be required for written English—is often helpful. They serve to set clauses off from one another. There is a tendency for a listener to lose track of the meaning of a sentence if the words run together. Read your sentence aloud while pretending to be a newscaster. The locations for additional commas should leap out at you.

The dash serves almost the same purpose as the comma, except that the dash does not cause the pitch to rise so severely. A rule of thumb is: Use dashes to divide phrases and commas to divide clauses.

Parentheses provide additional information to narrator's intonation function. They should be put around noun phrases of two or more content words. This means that the noun phrase, *a giant yacht* should be surrounded with parentheses because it contains two content words, *giant* and *yacht.* The phrase *my friend* should not have parentheses around it because it contains only one content word. Noun phrases can get fairly large, like *the best time I've ever had* or *a big basket of fruit and nuts.* The parentheses are most effective around these large phrases; the smaller ones can sometimes go without. The effect of parentheses is subtle, and in some sentences you might not notice their presence. In sentences of great length, however, they help provide for a very natural contour.

HINTS FOR INTELLIGIBILITY

There are a few tricks you can use to improve the intelligibility of a sentence. Often, a polysyllabic word is more recognizable than a monosyllabic word. For instance, instead of saying *huge*, say *enormous*. The longer version contains information in every syllable, thus giving the listener a greater chance to hear it correctly.

Another good practice is to keep sentences to an optimal length. Writing for reading and writing for speaking are two different things. Try not to write a sentence that cannot be easily spoken in one breath. Such a sentence tends to give the impression that the speaker has an infinite lung capacity and sounds unnatural. Try to keep sentences confined to one main idea; run-on sentences tend to lose their meaning.

New terms should be highly stressed the first time they are heard. This gives the listener something to cue on, and can aid in comprehension.

The insertion of the glottal stop phoneme Q at the end of a word can sometimes help prevent slurring of one word into another. When we speak, we do not pause at the end of each word, but instead transition smoothly between words. This can sometimes reduce intelligibility by eliminating word boundary cues. Placing a **Q**, (not the silent vowel **QX**) at the end of a word results in some phonological effects taking place which can restore the word boundary cues.

EXAMPLE OF ENGLISH AND PHONETIC TEXTS

Cardiomyopathy. I had never heard of it before, but there it was listed as the form of heart disease that felled not one or two but all three of the artificial heart recipients. A little research produced some interesting results. According to an article in the Nov. 8, 1984, *New England Journal of Medicine*, cigarette smoking causes this lethal disease that weakens the heart's pumping power. While the exact mechanism is not clear, Dr. Arthur J. Hartz speculated that nicotine or carbon monoxide in the smoke somehow poisons the heart and leads to heart failure.

KAA1RDIYOWMAYAA5PAXTHIY. AY /HAED NEH1VER /HER4D AXV IHT BIXFOH5R, BAHT DHEH5R IHT WAHZ - LIH4STIXD AEZ (DHAX FOH5RM AXV /HAA5RT DI-HZIY5Z) DHAET FEH4LD (NAAT WAH5N OHR TUW5) - BAHT (AO7L THRIY5 AXV DHAX AA5RTAXFIHSHUL /HAA5RTQ RIXSIH5PIYINTS). (AH LIH5TUL RIXSER5CH) PROHDUW5ST (SAHM IH5NTRIHSTIHNX RIXZAH5LTS). AHKOH5RDIHNX TUW (AEN AA5RTIHKUL IHN DHAX NOWVEH5MBER EY2TH NAY5NTIYNEYTIYFOH1R NUW IY5NXGLIND JER5NUL AXV MEH5DIXSIN), (SIH5GEREHT SMOW5KIHNX) KAO4ZIHZ (DHIHS LIY5THUL DIHZIY5Z) DHAET WIY4KINZ (DHAX /HAA5RTS PAH4MPIHNX PAW2ER). WAYL (DHIY IHGZAE5KT MEH5KINIXZUM) IHZ NAAT KLIY5R, DAA5KTER AA5RTHER JEY2 /HAARTS SPEH5KYULEYTIHD DHAET NIH5KAXTIYN, OHR KAA5RBIN MUNAA5KSAYD IHN DHAX SMOW5K - SAH5M/HAW1 POY4ZINZ DHAX /HAA5RT, AEND LIY4DZ TUW (/HAA5RT FEY5LYER).

CONCLUDING REMARKS

This guide should get you off to a good start in phonetic writing for Narrator. The only way to get really proficient is to practice. Many people become good at it in as little as one day. Others make continual mistakes because they find it hard to let go of the rules of English spelling, so trust your ears.

A More Technical Explanation

The narrator speech synthesizer is a computer model of the human speech production process. It attempts to produce accurately spoken utterances of any English sentence, given only a phonetic representation as input. Another program in the Amiga speech system, the translator device, derives the required phonetic spelling from English text. Timing and pitch contours are produced automatically by the synthesizer software.

In humans, the physical act of producing speech sounds begins in the lungs. To create a voiced sound, the lungs force air through the vocal folds (commonly called the vocal cords), which are held under tension and which periodically interrupt the flow of air, thus creating a buzz-like sound. This buzz, which has a spectrum rich in harmonics, then passes through the vocal tract and out the lips and nose, which alters its spectrum drastically. This is because the vocal tract acts as a frequency filter, selectively reinforcing some harmonics and suppressing others. It is this filtering that gives a speech sound its identity. The amplitude versus frequency graph of the filtering action is called the *vocal tract transfer function.* Changing the shape of the throat, tongue, and mouth retunes the filter system to accentuate different frequencies.

The sound travels as a pressure wave through the air, and it causes the listener's eardrum to vibrate. The ear and brain of the listener decode the incoming frequency pattern. From this the listener can subconsciously make a judgement about what physical actions were performed by the speaker to make the sound. Thus the speech chain is completed, the speaker having encoded his physical actions on a buzz via selective filtering and the listener having turned the sound into guesses about physical actions by frequency decoding.

Now that we know how humans produce speech, how does the Amiga do it? It turns out that the vocal tract transfer function is not random, but tends to accentuate energy in narrow bands called **formants**. The formant positions move fairly smoothly as we speak, and it is the formant frequencies to which our ears are sensitive. So, luckily, we do not have to model throat, tongue, teeth and lips with our computer, we can imitate formant actions instead.

A good representation of speech requires up to five formants, but only the lowest three are required for intelligibility. The pre-V37 Narrator had only three formants, while the V37 Narrator has five formants for a more natural sounding voice. We begin with an oscillator that produces a waveform similar to that which is produced by the vocal folds, and we pass it through a series of resonators, each tuned to a different formant frequency. By controlling the volume and pitch of the oscillator and the frequencies of the resonators, we can produce highly intelligible and natural-sounding speech. Of course the better the model the better the speech; but more importantly, experience has shown that the better the control of the model's parameters, the better the speech.

Oscillators, volume controls, and resonators can all be simulated mathematically in software, and it is by this method that the narrator system operates. The input phonetic string is converted into a series of target values for the various parameters. A system of rules then operates on the string to determine things such as the duration of each phoneme and the pitch contour. Transitions between target values are created and smoothed to produce natural, continuous changes from one sound to the next.

New values are computed for each parameter for every 8 milliseconds of speech, which produces about 120 acoustic changes per second. These values drive a mathematical model of the speech synthesizer. The accuracy of this simulation is quite good. Human speech has more formants that the narrator model, but they are high in frequency and low in energy content.

The human speech production mechanism is a complex and wonderful thing. The more we learn about it, the better we can make our computer simulations. Meanwhile, we can use synthetic speech as yet another computer output device to enhance the man/machine dialogue.

Example Speech and Mouth Movement Program

```
/*
 * Full_Narrator.c
 *
 * This example program sends a string of phonetic text to the narrator
 * device and, while it is speaking, highlights, word-by-word, a
 * corresponding English string.  In addition, mouth movements are drawn
 * in a separate window.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Requires Kickstart V37 or greater.
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <intuition/intuition.h>
#include <ctype.h>
#include <exec/exec.h>
#include <fcntl.h>
#include <devices/narrator.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/dos_protos.h>

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }      /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }   /* really */
#endif


/*
 *  Due to an omission, the sync field defines were not included in older
 *  versions of the narrator device include files.  So, if they haven't
 *  already been defined, do so now.
 */

#ifndef NDF_READMOUTH                           /* Already defined ? */
#define NDF_READMOUTH   0x01                    /* No, define here   */
#define NDF_READWORD    0x02
#define NDF_READSYL     0x04
#endif

#define PEN3    3                               /* Drawing pens */
#define PEN2    2
#define PEN1    1
#define PEN0    0


BOOL FromCLI = TRUE;
BYTE chans[4] = {3, 5, 10, 12};


LONG EyesLeft;                                  /* Left edge of left eye     */
LONG EyesTop;                                   /* Top of eyes box           */
LONG EyesBottom;                                /* Bottom of eyes box        */
LONG YMouthCenter;                              /* Pixels from top edge      */
LONG XMouthCenter;                              /* Pixels from left edge     */
LONG LipWidth, LipHeight;                       /* Width and height of mouth */


struct TextAttr MyFont = {"topaz.font", TOPAZ_SIXTY, FS_NORMAL, FPF_ROMFONT,};

struct  IntuitionBase   *IntuitionBase = NULL;
struct  GfxBase         *GfxBase = NULL;
struct  MsgPort         *VoicePort = NULL;
struct  MsgPort         *MouthPort = NULL;

struct  narrator_rb     *VoiceIO = NULL;
struct  mouth_rb        *MouthIO = NULL;

struct  IntuiText  HighLight;
struct  NewWindow  NewWindow;
struct  Window    *TextWindow;
struct  Window    *FaceWindow;
struct  RastPort  *FaceRast;


void main(int argc, char **argv)
{
LONG     i;
LONG     sentence;
LONG     Offset;
LONG     CharsLeft;
LONG     ScreenPos;
LONG     WordLength;
LONG     LineNum;
UBYTE    *Tempptr;
UBYTE    *English;
UBYTE    *OldEnglish;
UBYTE     c;

UBYTE    *PhonPtr;              /* Pointer to phonetic text     */
LONG     PhonSize;              /* Size of phonetic text        */
UBYTE    *PhonStart[100];       /* Start of phonetic sentences  */
LONG     NumPhonStarts;         /* Number of phonetic sentences */

UBYTE    *EngPtr;               /* Pointer to English text      */
LONG     EngSize;               /* Size of English text         */
UBYTE    *EngStart[100];        /* Start of English sentences   */
LONG     NumEngStarts;          /* Number of English sentences  */

UBYTE    *EngLine[24];          /* Start of line on screen      */
LONG     EngBytes[24];          /* Bytes per line on screen     */
LONG     NumEngLines;           /* Number of lines on screen    */


extern  void  Cleanup(UBYTE *errmsg);
extern  void  ClearWindow(struct Window *TextWindow);
extern  void  DrawFace(void);
extern  void  UpdateFace(void);


/*
 *  (0)   Note whether the program was started from the CLI or from
 *        Workbench.
 */

if (argc == 0)
    FromCLI = FALSE;

/*
 *  (1)   Setup the phonetic text to be spoken.  If there are any non-
 *        alphabetic characters in the text (such as NEWLINES or TABS)
 *        replace them with spaces.  Then break up the text into sentences,
 *        storing the start of each sentence in PhonStart array elements.
 */

PhonPtr = "KAA1RDIYOWMAYAA5PAXTHIY.  AY /HAED NEH1VER /HER4D AXV IHT "
          "BIXFOH5R, BAHT DHEH5R IHT WAHZ - LIH4STIXD AEZ (DHAX FOH5RM "
          "AXV /HAA5RT DIHZIY5Z) DHAET FEH4LD (NAAT WAH5N OHR TUW5) - "
          "BAHT (AO7L THRIY5 AXV DHAX AA5RTAXFIHSHUL /HAA5RTQ "
          "RIXSIH5PIYINTS).  (AH LIH5TUL RIXSER5CH) PROHDUW5ST (SAHM "
          "IH5NTRIHSTIHNX RIXZAH5LTS). AHKOH5RDIHNX TUW (AEN AA5RTIHKUL "
          "IHN DHAX NOWVEH5MBER EY2THQX NAY5NTIYNEYTIYFOH1R NUW IY5NXGLIND "
          "JER5NUL AXV MEH5DIXSIN), (SIH5GEREHT SMOW5KIHNX) KAO4ZIHZ "
          "(DHIHS LIY5THUL DIHZIY5Z) DHAET WIY4KINZ (DHAX /HAA5RTS "
          "PAH4MPIHNX PAW2ER).  WAYL (DHIY IHGZAE5KT MEH5KINIXZUM) IHZ "
          "NAAT KLIY5R, DAA5KTER AA5RTHER JEY2 /HAARTS SPEH5KYULEYTIHD "
          "DHAET NIH4KAXTIY2N- OHR KAA5RBIN MUNAA5KSAYD IHN DHAX SMOW5K- "
          "SAH5M/HAW1 POY4ZINZ DHAX /HAA5RT, AEND LIY4DZ TUW (/HAA5RT "
          "FEY5LYER).";
```

```
PhonSize = strlen(PhonPtr);
NumPhonStarts = 0;
PhonStart[NumPhonStarts++] = PhonPtr;
for (i = 0;  i < PhonSize;  ++i)
     {
     if (isspace((int)(c = *PhonPtr++)))
         *(PhonPtr-1) = ' ';
     if ((c == '.') || (c == '?'))
         {
         *PhonPtr = '\0';
         PhonStart[NumPhonStarts++] = ++PhonPtr;
         }
     }

/*
 *  (2)  Create the English text corresponding to the phonetic text above.
 *       As before, insure that there are no TABS or NEWLINES in the text.
 *       Break the text up into sentences and store the start of each
 *       sentence in EngStart array elements.
 */
EngPtr = "Cardiomyopathy. I had never heard of it before, but there it was "
         "listed as the form of heart disease that felled not one or two but "
         "all three of the artificial heart recipients. A little research "
         "produced some interesting results. According to an article in the "
         "November 8, 1984, New England Journal of Medicine, cigarette smoking "
         "causes this lethal disease that weakens the heart's pumping power.    "
         "While the exact mechanism is not clear, Doctor Arthur J Hartz "
         "speculated that nicotine or carbon monoxide in the smoke somehow "
         "poisons the heart and leads to heart failure.";

EngSize = strlen(EngPtr);
NumEngStarts = 0;
EngStart[NumEngStarts++] = EngPtr;
for (i = 0;  i < EngSize;  ++i)
     {
     if (isspace((int)(c = *EngPtr++)))
         *(EngPtr-1) = ' ';
     if ((c == '.') || (c == '?'))
         {
         *EngPtr = '\0';
         EngStart[NumEngStarts++] = ++EngPtr;
         }
     }

/*
 *  (3)   Open Intuition and Graphics libraries.
 */

if (!(IntuitionBase=(struct IntuitionBase *)OpenLibrary("intuition.library",0)))
    Cleanup("can't open intuition");

if ((GfxBase=(struct GfxBase *)OpenLibrary("graphics.library", 0)) == NULL)
    Cleanup("can't open graphics");

/*
 *  (4)   Setup the NewWindow structure for the text display and
 *        open the text window.
 */
NewWindow.LeftEdge    = 20;
NewWindow.TopEdge     = 100;
NewWindow.Width       = 600;
NewWindow.Height      = 80;
NewWindow.DetailPen   = 0;
NewWindow.BlockPen    = 1;
NewWindow.Title       = " Narrator Demo ";
NewWindow.Flags       = SMART_REFRESH | ACTIVATE | WINDOWDEPTH | WINDOWDRAG;
NewWindow.IDCMPFlags  = NULL;
NewWindow.Type        = WBENCHSCREEN;
NewWindow.FirstGadget = NULL;
NewWindow.CheckMark   = NULL;
NewWindow.Screen      = NULL;
NewWindow.BitMap      = NULL;
NewWindow.MinWidth    = 600;
NewWindow.MinHeight   = 80;
NewWindow.MaxWidth    = 600;
NewWindow.MaxHeight   = 80;
```

```
if ((TextWindow = (struct Window *)OpenWindow(&NewWindow)) == NULL)
    Cleanup("Text window could not be opened");

/*
 *  (4)   Setup the NewWindow structure for the face display, open the
 *        window, cache the RastPort pointer, and draw the initial face.
 */

NewWindow.LeftEdge    = 20;
NewWindow.TopEdge     = 12;
NewWindow.Width       = 120;
NewWindow.Height      = 80;
NewWindow.DetailPen   = 0;
NewWindow.BlockPen    = 1;
NewWindow.Title       = " Face ";
NewWindow.Flags       = SMART_REFRESH | WINDOWDEPTH | WINDOWDRAG;
NewWindow.IDCMPFlags  = NULL;
NewWindow.Type        = WBENCHSCREEN;
NewWindow.FirstGadget = NULL;
NewWindow.CheckMark   = NULL;
NewWindow.Screen      = NULL;
NewWindow.BitMap      = NULL;
NewWindow.MinWidth    = 120;
NewWindow.MinHeight   = 80;
NewWindow.MaxWidth    = 120;
NewWindow.MaxHeight   = 80;

if ((FaceWindow = (struct Window *)OpenWindow(&NewWindow)) == NULL)
    Cleanup("Face window could not be opened");

FaceRast = FaceWindow->RPort;

DrawFace();

/*
 *  (5)   Create read and write msg ports.
 */

if ((MouthPort = CreatePort(NULL,0)) == NULL)
    Cleanup("Can't get read port");
if ((VoicePort = CreatePort(NULL,0)) == NULL)
    Cleanup("Can't get write port");

/*
 *  (6)   Create read and write I/O request blocks.
 */

if (!(MouthIO = (struct mouth_rb *)
                  CreateExtIO(MouthPort,sizeof(struct mouth_rb))))
    Cleanup("Can't get read IORB");

if (!(VoiceIO = (struct narrator_rb *)
                   CreateExtIO(VoicePort,sizeof(struct narrator_rb))))
    Cleanup("Can't get write IORB");

/*
 *  (7)   Set up the write I/O request block and open the device.
 */

VoiceIO->ch_masks            = &chans[0];
VoiceIO->nm_masks            = sizeof(chans);
VoiceIO->message.io_Command = CMD_WRITE;
VoiceIO->flags               = NDF_NEWIORB;

if (OpenDevice("narrator.device", 0, VoiceIO, 0) != NULL)
    Cleanup("OpenDevice failed");

/*
 *  (8)   Set up the read I/O request block.
 */

MouthIO->voice.message.io_Device = VoiceIO->message.io_Device;
MouthIO->voice.message.io_Unit   = VoiceIO->message.io_Unit;
MouthIO->voice.message.io_Message.mn_ReplyPort = MouthPort;
MouthIO->voice.message.io_Command = CMD_READ;
```

```
/*
 *  (9)   Initialize highlighting IntuiText structure.
 */

HighLight.FrontPen  = 1;
HighLight.BackPen   = 0;
HighLight.DrawMode  = JAM1;
HighLight.ITextFont = &MyFont;
HighLight.NextText  = NULL;

/*
 *  (10)   For each sentence, put up the English text in BLACK.  As
 *         Narrator says each word, highlight that word in BLUE.  Also
 *         continuously draw mouth shapes as Narrator speaks.
 */

for (sentence = 0; sentence < NumPhonStarts; ++sentence)
     {
     /*
      *  (11)   Begin by breaking the English sentence up into lines of
      *         text in the window.  EngLine is an array containing a
      *         pointer to the start of each English text line.
      */

     English = EngStart[sentence] + strspn((UBYTE *)EngStart[sentence], " ");
     NumEngLines = 0;
     EngLine[NumEngLines++] = English;
     CharsLeft = strlen(English);
     while (CharsLeft > 51)
           {
           for (Offset = 51; *(English+Offset) != ' '; --Offset) ;
           EngBytes[NumEngLines-1] = Offset;
           English                += Offset + 1;
           *(English-1)            = '\0';
           EngLine[NumEngLines++]  = English;
           CharsLeft              -= Offset + 1;
           }
     EngBytes[NumEngLines-1] = CharsLeft;

     /*
      *  (12)   Clear the window and draw in the unhighlighted English text.
      */

     ClearWindow(TextWindow);

     HighLight.FrontPen = 1;
     HighLight.LeftEdge = 10;
     HighLight.TopEdge  = 20;

     for (i = 0; i < NumEngLines; ++i)
          {
          HighLight.IText = EngLine[i];
          PrintIText(TextWindow->RPort, &HighLight, 0, 0);
          HighLight.TopEdge += 10;
          }

     HighLight.TopEdge  = 20;
     HighLight.FrontPen = 3;
     HighLight.IText    = EngLine[0];

     /*
      *  (13)   Set up the write request with the address and length of
      *         the phonetic text to be spoken.  Also tell device to
      *         generate mouth shape changes and word sync events.
      */

     VoiceIO->message.io_Data   = PhonStart[sentence];
     VoiceIO->message.io_Length = strlen(VoiceIO->message.io_Data);
     VoiceIO->flags             = NDF_NEWIORB | NDF_WORDSYNC;
     VoiceIO->mouths            = 1;

    /*
     *  (14)   Send the write request to the device.  This is an
     *         asynchronous write, the device will return immediately.
     */

     SendIO(VoiceIO);
    /*
     *  (15)    Initialize some variables.
     */

     ScreenPos  = 0;
     LineNum    = 0;
     English    = EngLine[LineNum];
     OldEnglish = English;
     MouthIO->voice.message.io_Error = 0;

    /*
     *  (16)    Issue synchronous read requests.  For each request we
     *          check the sync field to see if the read returned a mouth
     *          shape change, a start of word sync event, or both.  We
     *          continue issuing read requests until we get a return code
     *          of ND_NoWrite, which indicates that the write has finished.
     */

     for (DoIO(MouthIO);MouthIO->voice.message.io_Error != ND_NoWrite;DoIO(MouthIO))
          {

          /*
           *  (17)    If bit 1 of the sync field is on, this is a start
           *          of word sync event.  In that case we highlight the
           *          next word.
           */

          if (MouthIO->sync & NDF_READWORD)
              {
              if ((Tempptr = strchr(English, ' ')) != NULL)
                  {
                  English = Tempptr + 1;
                  *(English-1) = '\0';
                  }
              PrintIText(TextWindow->RPort, &HighLight, 0, 0);
              WordLength      = strlen(OldEnglish) + 1;
              HighLight.IText = English;
              OldEnglish      = English;
              ScreenPos      += WordLength;

              if (ScreenPos >= EngBytes[LineNum])
                  {
                  HighLight.LeftEdge   = 10;
                  HighLight.TopEdge   += 10;
                  ScreenPos            = 0;
                  English = OldEnglish = EngLine[++LineNum];
                  HighLight.IText      = English;
                  }
              else
                  HighLight.LeftEdge += 10*WordLength;
              }

          /*
           *  (18)    If bit 0 of the sync field is on, this is a mouth
           *          shape change event.  In that case we update the face.
           */

          if (MouthIO->sync & NDF_READMOUTH)
              UpdateFace();


          }


     /*
      *  (19)    The write has finished (return code from last read equals
      *          ND_NoWrite).  We must wait on the write I/O request to
      *          remove it from the message port.
      */

     WaitIO(VoiceIO);

     }

/*
```

```
 *  (20)   Program completed, cleanup and return.
 */

Cleanup("Normal completion");

}


void Cleanup(UBYTE *errmsg)
{


/*
 *  (1)   Cleanup and go away.  This routine does not return but EXITs.
 *        Everything it does is pretty self explanatory.
 */

if (FromCLI)
    printf("%s\n\r", errmsg);
if (TextWindow)
    CloseWindow(TextWindow);
if (FaceWindow)
    CloseWindow(FaceWindow);
if (VoiceIO && VoiceIO->message.io_Device)
    CloseDevice(VoiceIO);
if (VoiceIO)
    DeleteExtIO(VoiceIO);
if (VoicePort)
    DeletePort(VoicePort);
if (MouthIO)
    DeleteExtIO(MouthIO);
if (MouthPort)
    DeletePort(MouthPort);
if (GfxBase)
    CloseLibrary(GfxBase);
if (IntuitionBase)
    CloseLibrary(IntuitionBase);

exit(RETURN_OK);
}


void ClearWindow(struct Window *TextWindow)
{
LONG    OldPen;

/*
 *  (1)   Clears a window.
 */

OldPen = (LONG)TextWindow->RPort->FgPen;
SetAPen(TextWindow->RPort, 0);
SetDrMd(TextWindow->RPort, JAM1);
RectFill(TextWindow->RPort, 3, 12, TextWindow->Width-3, TextWindow->Height-2);
SetAPen(TextWindow->RPort, OldPen);
}


void DrawFace()
{

/*
 *  (1)   Draws the initial face.  The variables defined here are used in
 *        UpdateFace() to redraw the mouth shape.
 */

EyesLeft = 15;
EyesTop = 20;
EyesBottom = 35;

XMouthCenter = FaceWindow->Width >> 1;
YMouthCenter = FaceWindow->Height - 25;

SetAPen(FaceWindow->RPort, PEN1);
RectFill(FaceWindow->RPort, 3, 10, FaceWindow->Width-3, FaceWindow->Height-2);

SetAPen(FaceWindow->RPort, PEN0);
```

```
RectFill(FaceWindow->RPort, EyesLeft, EyesTop, EyesLeft+25, EyesTop+15);
RectFill(FaceWindow->RPort, EyesLeft+65, EyesTop, EyesLeft+90, EyesTop+15);

SetAPen(FaceWindow->RPort, PEN3);
Move(FaceWindow->RPort, XMouthCenter-(FaceWindow->Width >> 3), YMouthCenter);
Draw(FaceWindow->RPort, XMouthCenter+(FaceWindow->Width >> 3), YMouthCenter);
}


void UpdateFace()
{
/*
 *  (1)   Redraws mouth shape in response to a mouth shape change message
 *        from the device.  Its all pretty self explanatory.
 */

WaitBOVP(&FaceWindow->WScreen->ViewPort);
SetAPen(FaceRast, PEN1);
RectFill(FaceRast, 3, EyesBottom, FaceWindow->Width-3, FaceWindow->Height-2);

LipWidth  = MouthIO->width*3;
LipHeight = MouthIO->height*2/3;

SetAPen(FaceRast, PEN3);
Move(FaceRast, XMouthCenter - LipWidth, YMouthCenter);
Draw(FaceRast, XMouthCenter           , YMouthCenter - LipHeight);
Draw(FaceRast, XMouthCenter + LipWidth, YMouthCenter);
Draw(FaceRast, XMouthCenter,            YMouthCenter + LipHeight);
Draw(FaceRast, XMouthCenter - LipWidth, YMouthCenter);
}
```

Additional Information on the Narrator Device

Additional programming information on the narrator device can be found in the include files and the Autodocs for the narrator device and the Autodocs for the translator library. All are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Narrator Device Information	
INCLUDES	devices/narrator.h devices/narrator.i
AUTODOCS	narrator.doc translator.doc

chapter nine
PARALLEL DEVICE

The parallel device provides a hardware-independent interface to the Amiga's Centronics-compatible parallel port. The primary use of the Amiga parallel port is for output to printers, but with its extensions for bi-directional I/O, it can also be used for communication with digitizers and high-speed links with other computers. The parallel device is based on the conventions of Exec device I/O, with extensions for parameter setting and control.

Parallel Device Commands and Functions

Command	Operation
CMD_FLUSH	Purge all queued requests for the parallel device. Does not affect active requests.
CMD_READ	Read a stream of characters from the parallel port. The number of characters can be specified or a termination character(s) can be used.
CMD_RESET	Reset the parallel port to its initialized state. All active and queued I/O requests will be aborted.
CMD_START	Restart all paused I/O over the parallel port. Reactivates the handshaking sequence.
CMD_STOP	Pause all active I/O over the parallel port. Deactivates the handshaking sequence.
CMD_WRITE	Write out a stream of characters to the parallel port. The number of characters can be specified or a NULL-terminated string can be sent.
PDCMD_QUERY	Return the status of the parallel port lines and registers.
PDCMD_SETPARAMS	Set the parameters of the parallel port.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the parallel device. If the command is in progress, it is stopped immediately. If it is queued, it is removed from the queue.
BeginIO()	Initiate a command and return immediately (asynchronous request). This is used to minimize the amount of system overhead.
CheckIO()	Determine the current state of an I/O request.
CloseDevice()	Relinquish use of the parallel device. All requests must be complete.
DoIO()	Initiate a command and wait for completion (synchronous request).
OpenDevice()	Obtain use of the parallel device.
SendIO()	Initiate a command and return immediately (asynchronous request).
WaitIO()	Wait for the completion of an asynchronous request. When the request is complete the message will be removed from your reply port.

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure of type **IOExtPar**. This structure will be used to communicate commands to the parallel device.
CreatePort()	Create a signal message port for reply messages from the parallel device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete an extended I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The parallel device operates like the other Amiga devices. To use it, you must first open the parallel device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The I/O request used by the parallel device is called **IOExtPar**.

```
struct   IOExtPar
    {
    struct  IOStdReq IOPar;
    ULONG   io_PExtFlags;          /* additional parallel flags */
    UBYTE   io_Status;             /* status of parallel port and registers */
    UBYTE   io_ParFlags;           /* parallel device flags */
    struct  IOPArray io_PTermArray; /* termination character array */
    };
```

See the include file *devices/parallel.h* for the complete structure definition.

OPENING THE PARALLEL DEVICE

Three primary steps are required to open the parallel device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an extended I/O request structure of type **IOExtPar** using **CreateExtIO()**. **CreateExtIO()** will initialize the I/O request to point to your reply port.
- Open the parallel device. Call **OpenDevice()**, passing the I/O request.

```
struct MsgPort  *ParallelMP;         /* Pointer to reply port */
struct IOExtPar *ParallelIO;         /* Pointer to I/O request */

if (ParallelMP=CreatePort(0,0) )
    if (ParallelIO=(struct IOExtPar *)
           CreateExtIO(ParallelMP,sizeof(struct IOExtPar)) )
        if (OpenDevice(PARALLELNAME,0L,(struct IORequest *)ParallelIO,0) )
            printf("%s did not open\n",PARALLELNAME);
```

During the open, the parallel device pays attention to just one flag; **PARF_SHARED**. For consistency, the other flag bits should also be properly set. Full descriptions of all flags will be given later. When the parallel device is opened, it fills the latest default parameter settings into the **IOExtPar** block.

READING FROM THE PARALLEL DEVICE

You read from the parallel device by passing an **IOExtPar** to the device with CMD_READ set in **io_Command**, the number of bytes to be read set in **io_Length** and the address of the read buffer set in **io_Data**.

```
#define READ_BUFFER_SIZE 256
char ParallelReadBuffer[READ_BUFFER_SIZE]; /* Reserve SIZE bytes of storage */

ParallelIO->IOPar.io_Length   = READ_BUFFER_SIZE;
ParallelIO->IOPar.io_Data     = (APTR)&ParallelReadBuffer[0];
ParallelIO->IOPar.io_Command  = CMD_READ;
DoIO((struct IORequest *)ParallelIO);
```

If you use this example, your task will be put to sleep waiting until the parallel device reads 256 bytes (or terminates early). Early termination can be caused by error conditions.

WRITING TO THE PARALLEL DEVICE

You write to the parallel device by passing an **IOExtPar** to the device with CMD_WRITE set in **io_Command**, the number of bytes to be written set in **io_Length** and the address of the write buffer set in **io_Data**.

To write a NULL-terminated string, set the length to -1; the device will output from your buffer until it encounters and transmits a value of zero (0x00).

```
ParallelIO->IOPar.io_Length   = -1;
ParallelIO->IOPar.io_Data     = (APTR)"Parallel lines cross 7 times... ";
ParallelIO->IOPar.io_Command  = CMD_WRITE;
DoIO((struct IORequest *)ParallelIO);          /* execute write */
```

The length of the request is -1, meaning we are writing a NULL-terminated string. The number of characters sent can be found in **io_Actual**.

CLOSING THE PARALLEL DEVICE

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**. When the last close is performed, the device will deallocate all resources and buffers. The latest parameter settings will be saved for the next open.

All I/O requests must be complete before **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()**:

```
if (!(CheckIO(ParallelIO)))
    {
    AbortIO(ParallelIO);  /* Ask device to abort request, if pending */
    }
WaitIO(ParallelIO);       /* Wait for abort, then clean up */
CloseDevice((struct IORequest *)ParallelIO);
```

Ending A Read or Write with Termination Characters

Reads and writes from the parallel device may terminate early if an error occurs or if an end-of-file is sensed. For example, if a break is detected on the line, any current read request will be returned with the error **ParErr_DetectedBreak**. The count of characters read to that point will be in the **io_Actual** field of the request.

You can specify a set of possible end-of-file characters that the parallel device is to look for in the input or output stream using the PDCMD_SETPARAMS command. These are contained in an **io_PTermArray** that you provide. **io_PTermArray** is used only when the PARF_EOFMODE flag is selected (see "Parallel Flags" below).

If EOF mode is selected, each input data character read into or written from the user's data block is compared against those in **io_PTermArray**. If a match is found, the **IOExtPar** is terminated as complete, and the count of characters transferred (including the termination character) is stored in **io_Actual**.

To keep this search overhead as efficient as possible, the parallel device requires that the array of characters be in descending order. The array has eight bytes and all must be valid (that is, do not pad with zeros unless zero is a valid EOF character). Fill to the end of the array with the lowest value termination character. When making an arbitrary choice of EOF character(s), you will get the quickest response from the lowest value(s) available.

```
/*
 * Terminate_Parallel.c
 *
 * This is an example of using a termination array for writes from the parallel
 * device. A termination array is set up for the characters Q, E, A and %.  The
 * EOFMODE flag is set in io_ParFlags to indicate that we want to use a
 * termination array by sending the PDCMD_SETPARAMS command to the device.
 * Then, a CMD_WRITE command is sent to the device with io_Length set to -1.
 *
 * The write will terminate when one of the four characters in the
 * termination array is sent or when the end of the write buffer has been reached.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/parallel.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void main(void)
{
struct MsgPort  *ParallelMP;          /* Define storage for one pointer */
struct IOExtPar *ParallelIO;         /* Define storage for one pointer */

struct IOPArray Terminators =
{
0x51454125,   /* Q E A A */
0x25252525    /* fill to end with lowest value, must be in descending order */
};
UBYTE *WriteBuffer ="abcdefghijklmEopqrstuvwxyze";
UWORD ctr;

if (ParallelMP=CreatePort(0,0))
    {
    if (ParallelIO=(struct IOExtPar *)
                CreateExtIO(ParallelMP,sizeof(struct IOExtPar)) )
        {
        if (OpenDevice(PARALLELNAME,0L,(struct IORequest *)ParallelIO,0) )
            printf("%s did not open\n",PARALLELNAME);
        else
            {
             /* Tell user what we are doing */
             printf("\fLooking for Q, E, A or % in output\n");

             /* Set EOF mode flag
              * Set the termination array
              * Send PDCMD_SETPARAMS to the parallel device
              */
             ParallelIO->io_ParFlags |= PARF_EOFMODE;
             ParallelIO->io_PTermArray = Terminators;
             ParallelIO->IOPar.io_Command  = PDCMD_SETPARAMS;
             if (DoIO((struct IORequest *)ParallelIO))
                 printf("Set Params failed ");   /* Inform user of error */
             else
                 {
                 /* Send buffer */
                 ParallelIO->IOPar.io_Length   = -1;
                 ParallelIO->IOPar.io_Data     = WriteBuffer;
                 ParallelIO->IOPar.io_Command  = CMD_WRITE;
                 if (DoIO((struct IORequest *)ParallelIO))
                     printf("Error: Write failed\n");
                 else
                     {
                      /* Display all characters sent */
                      printf("\nThese characters were sent:\n\t\t\tASCII\tHEX\n");
                      for (ctr=0;ctr<ParallelIO->IOPar.io_Actual;ctr++)
                           printf("\t\t\t%c\t%x\n",*WriteBuffer,*WriteBuffer++);
                      printf("\nThe actual number of characters sent: %d\n",
                                ParallelIO->IOPar.io_Actual);
                      }
                 }
            CloseDevice((struct IORequest *)ParallelIO);
            }

        DeleteExtIO((struct IORequest *)ParallelIO);
        }
    else
        printf("Error: Could not create I/O request\n");

    DeletePort(ParallelMP);
    }
else
    printf("Error: Could not create message port\n");
}
```

The read will terminate before the **io_Length** number of characters is read if a ‘Q’, ‘E’, or ‘A’ is detected.

It's Usually For Output. Most applications for the parallel device use the device for output, hence the termination feature is usually done on the output stream.

Setting Parallel Parameters

You can control the parallel parameters shown in the following table. The parameter name within the parallel **IOExtPar** data structure is shown below. All of the fields described in this section are filled with defaults when you call **OpenDevice()**. Thus, you need not worry about any parameter that you do not need to change. The parameters are defined in the include file *devices/parallel.h.*

Parallel Parameters (IOExtPar)

IOExtPar Field Name	Parallel Device Parameter It Controls
io_PExtFlags	Reserved for future use.
io_PTermArray	A byte-array of eight termination characters, must be in descending order. If EOFMODE is set in the parallel flags, this array specifies eight possible choices of characters to use as an end-of-file mark. See the section above titled "Ending A Read Or Write with Termination Characters" and the PDCMD_SETPARAMS summary page in the Autodocs.
io_Status	Contains status information. It is filled in by the PDCMD_QUERY command.
io_ParFlags	See "Parallel Flags" below.

You set the parallel parameters by passing an **IOExtPar** to the device with PDCMD_SETPARAMS set in **io_Command** and with the flags and parameters set to the values you want.

```
ParallelIO->io_ParFlags      &= ~PARF_EOFMODE;      /* Set EOF mode */
ParallelIO->IOPar.io_Command  =  PDCMD_SETPARAMS;   /* Set params command */
if (DoIO(ParallelIO);
    printf("Error setting parameters!\n");
```

The above code fragment modifies one bit in **io_ParFlags**, then sends the command.

Proper Time for Parameter Changes. A parameter change should not be performed while an I/O request is actually being processed, because it might invalidate already active request handling. Therefore you should use PDCMD_SETPARAMS only when you have no parallel I/O requests pending.

PARALLEL FLAGS (bit definitions for io_ParFlags)

The flags shown in the following table can be set to affect the operation of the parallel device. Note that the default state of all of these flags is zero. The flags are defined in the include file *devices/parallel.h.*

Parallel Flags (io_ParFlags)

Flag Name	Effect on Device Operation
PARF_EOFMODE	Set this bit if you want the parallel device to check I/O characters against **io_TermArray** and terminate the I/O request immediately if an end-of-file character has been encountered. *Note*: This bit can be set and reset directly in the user's **IOExtPar** block without a call to PDCMD_SETPARAMS.
PARF_ACKMODE	Set this bit if you want to use ACK handshaking.
PARF_FASTMODE	Set this bit if you want to use high-speed mode for transfers to high-speed printers. This mode will send out data as long as the BUSY signal is low. The printer must be able to raise the BUSY signal within three microseconds or data will be lost. Should only be used when the device has been opened for exclusive-access.
PARF_SLOWMODE	Set this bit if you want to use slow-speed mode for transfers to very slow printers. Should not be used with high-speed printers.
PARF_SHARED	Set this bit if you want to allow other tasks to simultaneously access the parallel port. The default is exclusive access. If someone already has the port, whether for exclusive or shared access, and you ask for exclusive access, your **OpenDevice()** call will fail (must be modified before **OpenDevice()**).

Querying the Parallel Device

You query the parallel device by passing an **IOExtPar** to the device with PDCMD_QUERY set in **io_Command**. The parallel device will respond with the status of the parallel port lines and registers.

```
UWORD Parallel_Status;

ParallelIO->IOPar.io_Command  = PDCMD_QUERY; /* indicate query */
DoIO((struct IORequest *)ParallelIO);

Parallel_Status = ParallelIO->io_Status; /* store returned status */
```

The 8 status bits of the parallel device are returned in **io_Status**.

Parallel Device Status Bits

Bit	Active	Function
0	high	Printer busy toggle (offline)
1	high	Paper out
2	high	Printer Select on the A1000. On the A500 and A2000, select is also connected to to the parallel port's Ring Indicator. Be cautious when making cables.
3	—	read=0; write=1
4–7	—	(reserved)

The parallel device also returns error codes whenever an operation is attempted.

```
struct IOPArray Terminators =
{
0x51454141,    /* Q E A A */
0x41414141     /* fill to end with lowest value, must be in descending order */
};

ParallelIO->io_ParFlags != PARF_EOFMODE;          /* Set EOF mode flag */
ParallelIO->io_PTermArray = Terminators;         /* Set termination characters */
ParallelIO->IOPar.io_Command  = PDCMD_SETPARAMS; /* Set parameters */
if (DoIO((struct IORequest *)ParallelIO))
    printf("Set Params failed. Error: %d ",ParallelIO->IOPar.io_Error);
```

The error is returned in the **io_Error** field of the **IOExtPar** structure.

Parallel Device Error Codes

Error	Value	Explanation
ParErr_DevBusy	1	Device in use
ParErr_BufToBig	2	Out of memory
ParErr_InvParam	3	Invalid parameter
ParErr_LineErr	4	Parallel line error
ParErr_NotOpen	5	Device not open
ParErr_PortReset	6	Port Reset
ParErr_InitErr	7	Initialization Error

```
/*
 * Parallel.c
 *
 * Parallel device example
 *
 * Compile with SAS C 5.10: LC -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <devices/parallel.h>
```

```
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void main(void)
{
struct MsgPort  *ParallelMP;          /* Define storage for one pointer */
struct IOExtPar *ParallelIO;         /* Define storage for one pointer */
ULONG            WaitMask;          /* Collect all signals here       */
ULONG            Temp;             /* Hey, we all need pockets :-)   */

if (ParallelMP=CreatePort(0,0) )
    {
    if (ParallelIO=(struct IOExtPar *)
        CreateExtIO(ParallelMP,sizeof(struct IOExtPar)) )
        {
        if (OpenDevice(PARALLELNAME,0L,(struct IORequest *)ParallelIO,0) )
            printf("%s did not open\n",PARALLELNAME);
        else
            {
            /* Precalculate a wait mask for the CTRL-C, CTRL-F and message port
             * signals.  When one or more signals are received, Wait() will
             * return.  Press CTRL-C to exit the example.  Press CTRL-F to
             * wake up the example without doing anything. NOTE: A signal
             * may show up without an associated message!
             */
            WaitMask = SIGBREAKF_CTRL_C | SIGBREAKF_CTRL_F |
                       1L << ParallelMP->mp_SigBit;

            ParallelIO->IOPar.io_Command  = CMD_WRITE;
            ParallelIO->IOPar.io_Length   = -1;
            ParallelIO->IOPar.io_Data     = (APTR)"Hey, Jude!\\r\n";
            SendIO(ParallelIO);             /* execute write */

            printf("Sleeping until CTRL-C, CTRL-F, or write finish\n");
            while(1)
                {
                Temp = Wait(WaitMask);
                printf("Just woke up (YAWN!)\n");

                if (SIGBREAKF_CTRL_C & Temp)
                    break;

                if (CheckIO(ParallelIO) ) /* If request is complete... */
                    {
                    WaitIO(ParallelIO);   /* clean up and remove reply */
                    printf("%ld bytes sent\n",ParallelIO->IOPar.io_Actual);
                    break;
                    }
                }

            AbortIO(ParallelIO);  /* Ask device to abort request, if pending */
            WaitIO(ParallelIO);   /* Wait for abort, then clean up */

            CloseDevice((struct IORequest *)ParallelIO);
            }
        DeleteExtIO(ParallelIO);
        }
    DeletePort(ParallelMP);
    }
}
```

Additional Information on the Parallel Device

Additional programming information on the parallel device can be found in the include files and the Autodocs for the parallel device. Both are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Parallel Device Information	
INCLUDES	devices/parallel.h
	devices/parallel.i
AUTODOCS	parallel.doc

chapter ten

PRINTER DEVICE

The printer device offers a way of sending configuration-independent output to a printer attached to the Amiga. It can be thought of as a filter: it takes standard commands as input and translates them into commands understood by the printer. The commands sent to the printer are defined in a specific printer driver program. For each type of printer in use, a driver (or the driver of a compatible printer) should be present in the *devs:printers* directory.

Printer Driver Source Code In This Chapter	
EpsonX	A YMCB, 8 pin, multi-density interleaved printer.
HP_LaserJet	A black and white, multi-density, page-oriented printer.

Printer Device Commands and Functions

Command	**Operation**
CMD_FLUSH	Remove all queued requests for the printer device. Does not affect active requests.
CMD_RESET	Reset the printer device to its initialized state. All active and queued I/O requests will be aborted.
CMD_START	Restart all paused I/O requests
CMD_STOP	Pause all active and queued I/O requests.
CMD_WRITE	Write out a stream of characters to the printer device. The number of characters can be specified or a NULL-terminated string can be sent.
PRD_DUMPRPORT	Dump the specified **RastPort** to a graphics printer.
PRD_PRTCOMMAND	Send a command to the printer.
PRD_QUERY	Return the status of the printer port's lines and registers.
PRD_RAWWRITE	Send unprocessed output to the the printer.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the printer device.
CloseDevice()	Relinquish use of the printer device. All requests must be complete before closing.
DoIO()	Start a command and wait for completion (synchronous request).
OpenDevice()	Obtain use of the printer device.
SendIO()	Start a command and return immediately (asynchronous request).
WaitIO()	Wait for the completion of an asynchronous request. When the request is complete, the message will be removed from the printer message port.

Exec Support Functions as Used in This Chapter

CreatePort()	Create a signal message port for reply messages from the audio device. Exec will signal a task when a message arrives at the reply port.
CreateExtIO()	Create an I/O request structure of type **printerIO**. This structure will be used to send commands to the printer device.
DeletePort()	Delete the message port created by **CreatePort()**.
DeleteExtIO()	Delete an I/O request structure created by **CreateExtIO()**.

Printer Device Access

The printer device is totally transparent to an application. It uses information set up by the Workbench Preferences Printer and PrinterGfx tools to identify the type of printer connection (serial or parallel), type of dithering, etc. It also offers the flexibility to send raw information to the printer for special non-standard or unsupported features. Raw data transfer is not recommended for conventional text and graphics since it will result in applications that will only work with certain printers. By using the standard printer device interface, an application can perform device independent output to a printer.

> *Don't Hog The Device.* The printer device is currently an exclusive access device. Do not tie it up needlessly.

There are two ways of doing output to the printer device:

- **PRT:—the AmigaDOS printer device**
  PRT: may be opened just like any other AmigaDOS file. You may send standard escape sequences to PRT: to specify the options you want as shown in the command table below. The escape sequences are interpreted by the printer driver, translated into printer-specific escape sequences and forwarded to the printer. When using PRT: the escape sequences and data must be sent as a character stream. Using PRT: is by far the easiest way of doing text output to a printer.

- **printer.device—to directly access the printer device itself**
  By opening the printer device directly, you have full control over the printer. You can either send standard escape sequences as shown in the command table below or send raw characters directly to the printer with no processing at all. Doing this would be similar to sending raw characters to SER: or PAR: from AmigaDOS. (Since this interferes with device-independence it is strongly discouraged). Direct access to the printer device also allows you to transmit device I/O commands, such as reset and flush, and do a raster dump on a graphics-capable printer.

  Use A Stream to Escape. All "raw escape sequences" transmitted to the printer through the printer device must take the form of a character stream.

OPENING PRT:

When using the printer device as PRT:, you can open it just as though it were a normal AmigaDOS output file.

```
struct FileHandle *file;

file = Open( "PRT:", MODE_NEWFILE );     /* Open PRT: */
if (file == 0)                           /* if the open was unsuccessful */
    exit(PRINTER_WONT_OPEN);
```

WRITING TO PRT:

Once you've opened it, you can print by calling the AmigaDOS **Write()** standard I/O routine.

```
actual_length = Write(file, dataLocation, length);
```

where

file
is a file handle.

dataLocation
is a pointer to the first character in the output stream you wish to write. This stream can contain the standard escape sequences as shown in the command table below. The printer command aRAW (see the Printer Device Command Functions table below) can be used in the stream if character translation is not desired.

length
is the length of the output stream.

actual_length
is the actual length of the write. For the printer device, if there are no errors, this will be the same as the length of write requested. The only exception is if you specify a value of -1 for length. In this case, -1 for length means that a null (0) terminated stream is being written to the printer device. The device returns the count of characters written prior to encountering the null. If it returns a value of -1 in **actual_length**, there has been an error.

-1 = STOP! If a -1 is returned by **Write()**, do not do any additional printing.

CLOSING PRT:

When the printer I/O is complete, you should close PRT:. Don't keep the device open when you are not using it. The user may have changed the printer settings by using the Workbench Preferences tool. There's also the possibility the printer has been turned off and on again causing the printer to switch to its own default settings. Every time the printer device is opened, it reads the current Preferences settings. Hence, by always opening the printer device just before printing and always closing it afterwards, you ensure that your application is using the current Preferences settings.

```
Close(file);
```

In DOS, You Must Be A Process. Printer I/O through the DOS must be done by a process, *not* by a task. DOS utilizes information in the process control block and would become confused if a simple task attempted to perform these activities. Printer I/O using the printer device directly, however, *can* be performed by a task.

The remainder of this chapter will deal with using the printer device directly.

Device Interface

The printer device operates like the other Amiga devices. To use it, you must first open the printer device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga Devices" chapter for general information on device usage.

There are three distinct kinds of data structures required by the printer I/O routines. Some of the printer device I/O commands, such as CMD_START and CMD_WRITE require only an **IOStdReq** data structure. Others, such as PRD_DUMPRPORT and PRD_PRTCOMMAND, require an extended data structure called **IODRPReq** (for "Dump a RastPort Request") or **IOPrtCmdReq** (for "Printer Command Request").

For convenience, it is strongly recommended that *you define* a single data structure called **printerIO**, that can be used to represent any of the three pre-defined printer communications request blocks.

```
union printerIO
{
    struct IOStdReq    ios;
    struct IODRPReq    iodrp;
    struct IOPrtCmdReq iopc;
};

struct IODRPReq
{
    struct  Message io_Message;
    struct  Device  *io_Device;      /* device node pointer  */
    struct  Unit    *io_Unit;        /* unit (driver private)*/
    UWORD   io_Command;              /* device command */
    UBYTE   io_Flags;
    BYTE    io_Error;                /* error or warning num */
    struct  RastPort *io_RastPort;   /* raster port */
    struct  ColorMap *io_ColorMap;   /* color map */
    ULONG   io_Modes;                /* graphics viewport modes */
    UWORD   io_SrcX;                 /* source x origin */
    UWORD   io_SrcY;                 /* source y origin */
    UWORD   io_SrcWidth;             /* source x width */
    UWORD   io_SrcHeight;            /* source x height */
    LONG    io_DestCols;             /* destination x width */
    LONG    io_DestRows;             /* destination y height */
    UWORD   io_Special;              /* option flags */
};

struct IOPrtCmdReq
{
    struct  Message io_Message;
    struct  Device  *io_Device;      /* device node pointer  */
    struct  Unit    *io_Unit;        /* unit (driver private)*/
    UWORD   io_Command;              /* device command */
    UBYTE   io_Flags;
    BYTE    io_Error;                /* error or warning num */
    UWORD   io_PrtCommand;           /* printer command */
    UBYTE   io_Parm0;                /* first command parameter */
    UBYTE   io_Parm1;                /* second command parameter */
    UBYTE   io_Parm2;                /* third command parameter */
    UBYTE   io_Parm3;                /* fourth command parameter */
};
```

See the include file *exec/io.h* for more information on **IOStdReq** and the include file *devices/printer.h* for more information on **IODRPReq** and **IOPrtCmdReq**.

OPENING THE PRINTER DEVICE

Three primary steps are required to open the printer device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an extended I/O request structure of type **printerIO** with the **CreateExtIO()** function. This means that one memory area can be used to represent three distinct forms of memory layout for the three different types of data structures that must be used to pass commands to the printer device. By using **CreateExtIO()**, you automatically allocate enough memory to hold the largest structure in the union statement.
- Open the printer device. Call **OpenDevice()**, passing the I/O request.

```
union printerIO
{
    struct IOStdReq    ios;
    struct IODRPReq    iodrp;
    struct IOPrtCmdReq iopc;
};

struct MsgPort  *PrintMP;          /* Message port pointer */
union printerIO *PrintIO;          /* I/O request pointer */

if (PrintMP=CreateMsgPort() )
    if (PrintIO=(union printerIO *)
                CreateExtIO(PrintMP,sizeof(union printerIO)) )
        if (OpenDevice("printer.device",0L,(struct IORequest *)PrintIO,0) )
            printf("printer.device did not open\n");
```

The printer device automatically fills in default settings for all printer device parameters from Preferences. In addition, information about the printer itself is placed into the appropriate fields of **printerIO**. (See the "Obtaining Printer Specific Data" section below.)

> *Pre-V36 Tasks and OpenDevice().* Tasks in pre-V36 versions of the operating system are not able to safely **OpenDevice()** the printer device because it may be necessary to load it in from disk, something only a process could do under pre-V36. V36 and higher versions of the operating system do not have such a limitation.

WRITING TEXT TO THE PRINTER DEVICE

Text written to a printer can be either processed text or unprocessed text.

Processed text is written to the device using the CMD_WRITE command. The printer device accepts a character stream, translates any embedded escape sequences into the proper sequences for the printer being used and then sends it to the printer. The escape sequence translation is based on the printer driver selected either through Preferences or through your application. You may also send a NULL-terminated string as processed text.

Unprocessed text is written to the device using the PRD_RAWWRITE command. The printer device accepts a character stream and sends it unchanged to the printer. This implies that you know the exact escape sequences required by the printer you are using. You may not send a NULL-terminated string as unprocessed text.

One additional point to keep in mind when using PRD_RAWWRITE is that Preference settings for the printer are ignored. Unless the printer has already been initialized by another command, the printer's own default settings will be used when printing raw, *not the user's Preferences settings.*

You write processed text to the printer device by passing an **IOStdReq** to the device with CMD_WRITE set in **io_Command**, the number of bytes to be written set in **io_Length** and the address of the write buffer set in **io_Data**.

To write a NULL-terminated string, set the length to -1; the device will output from your buffer until it encounters a value of zero (0x00).

```
PrintIO->ios.io_Length   = -1;
PrintIO->ios.io_Data     = (APTR)"I went to a fight and a hockey game broke out."
PrintIO->ios.io_Command  = CMD_WRITE;
DoIO((struct IORequest *)PrintIO);
```

The length of the request is -1, meaning we are writing a NULL-terminated string. The number of characters sent will be found in **io_Actual** after the write request has completed.

You write unprocessed text to the printer device by passing an **IOStdReq** to the device with PRD_RAWWRITE set in **io_Command**, the number of bytes to be written set in **io_Length** and the address of the write buffer set in **io_Data**.

```
UBYTE *outbuffer;

PrintIO->ios.io_Length   = strlen(outbuffer);
PrintIO->ios.io_Data     = (APTR)outbuffer;
PrintIO->ios.io_Command  = PRD_RAWWRITE;
DoIO((struct IORequest *)PrintIO);
```

IOStdReq Only. I/O requests with CMD_WRITE and PRD_RAWWRITE must use the **IOStdReq** structure of the union **printerIO**.

IMPORTANT POINTS ABOUT PRINT REQUESTS

- **Perform printer I/O from a separate task or process**
  It is quite reasonable for a user to expect that printing will be performed as a background operation. You should try to accommodate this expectation as much as possible.
- **Give the user a chance to stop**
  Your application should always allow the user to stop a print request before it is finished.
- **Don't confuse aborting a print request with cancelling a page**
  Some applications seem to offer the user the ability to abort a multi-page print request when in fact the abort is only for the current page being printed. This results in the next page being printed instead of the request being stopped. **Do not do this!** It only confuses the user and takes away from your application. There is nothing wrong with allowing the user to cancel a page and continue to the next page, but it should be explicit that this is the case. If you abort a print request, the entire request should be aborted.

CLOSING THE PRINTER DEVICE

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**.

All I/O requests must be complete before **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()**.

```
AbortIO(PrintIO);  /* Ask device to abort request, if pending */
WaitIO(PrintIO);   /* Wait for abort, then clean up */

CloseDevice((struct IORequest *)PrintIO);
```

> *Use AbortIO()/WaitIO() Intelligently.* Only call **AbortIO()/WaitIO()** for requests which have already been *sent* to the printer device. Using the **AbortIO()/WaitIO()** sequence on requests which have not been sent results in a hung condition.

Sending Printer Commands to a Printer

As mentioned before, it is possible to include printer commands (escape sequences) in the character stream and send them to the printer using the CMD_WRITE device I/O command. It is also possible to use the *printer command names* using the device I/O command PRD_PRTCOMMAND with the **IOPrtCmdReq** data structure. This gives you a mnemonic way of setting the printer to your program needs.

You send printer commands to the device by passing an **IOPrtCmdReq** to the device with PRD_PRTCOMMAND set in **io_Command**, the printer command set in **io_PrtCommand** and up to four parameters set in **Parm0** through **Parm3**.

```
#include <devices/printer.h>

PrintIO->iopc.io_PrtCommand = aSLRM;  /* Set left & right margins */
PrintIO->iopc.io_Parm0 = 1;           /* Set left margin = 1 */
PrintIO->iopc.io_Parm1 = 79;          /* Set right margin = 79 */
PrintIO->iopc.io_Parm2 = 0;
PrintIO->iopc.io_Parm3 = 0;
PrintIO->iopc.io_Command = PRD_PRTCOMMAND;
DoIO((struct IORequest *)PrintIO);
```

Consult the command function table listed below for other printer commands.

PRINTER COMMAND DEFINITIONS

The following table describes the supported printer functions.

> *Just Because We Have It Doesn't Mean You Do.* Not all printers support every command. Unsupported commands will either be ignored or simulated using available functions.

To transmit a command to the printer device, you can either formulate a character stream containing the material shown in the "Escape Sequence" column of the table below or send an PRD_PRTCOMMAND device I/O command to the printer device with the "Name" of the function you wish to perform.

Printer Device Command Functions

Name	Cmd No.	Escape Sequence	Function	Defined by:
aRIS	0	ESCc	Reset	ISO
aRIN	1	ESC#1	Initialize	+++
aIND	2	ESCD	Linefeed	ISO
aNEL	3	ESCE	Return,linefeed	ISO
aRI	4	ESCM	Reverse linefeed	ISO
aSGR0	5	ESC[0m	Normal char set	ISO
aSGR3	6	ESC[3m	Italics on	ISO
aSGR23	7	ESC[23m	Italics off	ISO
aSGR4	8	ESC[4m	Underline on	ISO
aSGR24	9	ESC[24m	Underline off	ISO
aSGR1	10	ESC[1m	Boldface on	ISO
aSGR22	11	ESC[22m	Boldface off	ISO
aSFC	12	ESC[nm	Set foreground color where *n* stands for a pair of ASCII digits, 3 followed by any number 0–9 (See ISOColor Table)	ISO
aSBC	13	ESC[nm	Set background color where *n* stands for a pair of ASCII digits, 4 followed by any number 0–9 (See ISO Color Table)	ISO
aSHORP0	14	ESC[0w	Normal pitch	DEC
aSHORP2	15	ESC[2w	Elite on	DEC
aSHORP1	16	ESC[1w	Elite off	DEC
aSHORP4	17	ESC[4w	Condensed fine on	DEC
aSHORP3	18	ESC[3w	Condensed off	DEC
aSHORP6	19	ESC[6w	Enlarged on	DEC
aSHORP5	20	ESC[5w	Enlarged off	DEC
aDEN6	21	ESC[6"z	Shadow print on	DEC (sort of)
aDEN5	22	ESC[5"z	Shadow print off	DEC
aDEN4	23	ESC[4"z	Doublestrike on	DEC
aDEN3	24	ESC[3"z	Doublestrike off	DEC
aDEN2	25	ESC[2"z	NLQ on	DEC
aDEN1	26	ESC[1"z	NLQ off	DEC
aSUS2	27	ESC[2v	Superscript on	+++
aSUS1	28	ESC[1v	Superscript off	+++
aSUS4	29	ESC[4v	Subscript on	+++
aSUS3	30	ESC[3v	Subscript off	+++
aSUS0	31	ESC[0v	Normalize the line	+++
aPLU	32	ESCL	Partial line up	ISO
aPLD	33	ESCK	Partial line down	ISO
aFNT0	34	ESC(B	US char set or Typeface 0	DEC
aFNT1	35	ESC(R	French char set or Typeface 1	DEC

aFNT2	36	ESC(K	German char set or Typeface 2	DEC
aFNT3	37	ESC(A	UK char set or Typeface 3	DEC
aFNT4	38	ESC(E	Danish I char set or Typeface 4	DEC
aFNT5	39	ESC(H	Swedish char set or Typeface 5	DEC
aFNT6	40	ESC(Y	Italian char set or Typeface 6	DEC
aFNT7	41	ESC(Z	Spanish char set or Typeface 7	DEC
aFNT8	42	ESC(J	Japanese char set or Typeface 8	+++
aFNT9	43	ESC(6	Norwegian char set or Typeface 9	DEC
aFNT10	44	ESC(C	Danish II char set or Typeface 10 (See Suggested Typefaces Table)	+++
aPROP2	45	ESC[2p	Proportional on	+++
aPROP1	46	ESC[1p	Proportional off	+++
aPROP0	47	ESC[0p	Proportional clear	+++
aTSS	48	ESC[n E	Set proportional offset	ISO
aJFY5	49	ESC[5 F	Auto left justify	ISO
aJFY7	50	ESC[7 F	Auto right justify	ISO
aJFY6	51	ESC[6 F	Auto full justify	ISO
aJFY0	52	ESC[0 F	Auto justify off	ISO
aJFY3	53	ESC[3 F	Letter space (justify)	ISO (special)
aJFY1	54	ESC[1 F	Word fill(auto center)	ISO (special)
aVERP0	55	ESC[0z	1/8" line spacing	+++
aVERP1	56	ESC[1z	1/6" line spacing	+++
aSLPP	57	ESC[nt	Set form length n	DEC
aPERF	58	ESC[nq	Perf skip n (n>0)	+++
aPERF0	59	ESC[0q	Perf skip off	+++
aLMS	60	ESC#9	Left margin set	+++
aRMS	61	ESC#0	Right margin set	+++
aTMS	62	ESC#8	Top margin set	+++
aBMS	63	ESC#2	Bottom margin set	+++
aSTBM	64	ESC[n;nr	Top and bottom margins	DEC
aSLRM	65	ESC[n;ns	Left and right margins	DEC
aCAM	66	ESC#3	Clear margins	+++
aHTS	67	ESCH	Set horizontal tab	ISO
aVTS	68	ESCJ	Set vertical tabs	ISO
aTBC0	69	ESC[0g	Clear horizontal tab	ISO
aTBC3	70	ESC[3g	Clear all h. tabs	ISO
aTBC1	71	ESC[1g	Clear vertical tab	ISO
aTBC4	72	ESC[4g	Clear all v. tabs	ISO
aTBCALL	73	ESC#4	Clear all h. & v. tabs	+++
aTBSALL	74	ESC#5	Set default tabs	+++
aEXTEND	75	ESC[n"x	Extended commands	+++
aRAW	76	ESC[n"r	Next n chars are raw	+++

Legend:

ISO	indicates that the sequence has been defined by the International Standards Organization. This is also very similar to ANSI x3.64.
DEC	indicates a control sequence defined by Digital Equipment Corporation.
+++	indicates a sequence unique to Amiga.
n	stands for a decimal number expressed as a set of ASCII digits. In the aRAW string ESC[5"rHELLO, n is substituted by 5, the number of RAW characters you send to the printer.

ISO Color Table

0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	NC
9	Default

Suggested Typefaces

0	Default typeface
1	Line Printer or equivalent
2	Pica or equivalent
3	Elite or equivalent
4	Helvetica or equivalent
5	Times Roman or equivalent
6	Gothic or equivalent
7	Script or equivalent
8	Prestige or equivalent
9	Caslon or equivalent
10	Orator or equivalent

Obtaining Printer Specific Data

Information about the printer in use can be obtained by reading the **PrinterData** and **PrinterExtendedData** structures. The values found in these structures are determined by the pr inter driver selected through Preferences. The data structures are defined in *devices/prtbase.h.*

Printer specific data is returned in **printerIO** when the printer device is opened. To read the structures, you must first set the **PrinterData** structure to point to **iodrp.io_Device** of the **printerIO** used to open the device and then set **PrinterExtendedData** to point to the extended data portion of **PrinterData**.

```
/*
 * Printer_Data.c
 *
 * Example getting driver specifics.
 *
 * Compiled with SAS C 5.10a. lc -cfist -v -L Printer_Data
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/ports.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>
```

```
union printerIO
{
    struct IOStdReq    ios;
    struct IODRPReq    iodrp;
    struct IOPrtCmdReq iopc;
};

VOID main(VOID);

VOID main(VOID)
{
struct MsgPort  *PrinterMP;
union printerIO *PIO;
struct PrinterData *PD;
struct PrinterExtendedData *PED;

/* Create non-public messageport. Could use CreateMsgPort() for this, that's
 * V37 specific however.
 */
if (PrinterMP = (struct MsgPort *)CreatePort(0,0))
    {
    /* Allocate printerIO union */
    if (PIO = (union printerIO *)CreateExtIO(PrinterMP, sizeof(union printerIO)))
        {
        /* Open the printer.device */
        if (!(OpenDevice("printer.device",0,(struct IORequest *)PIO,0)))
            {

            /* Now that we've got the device opened, let's see what we've got.
             * First, get a pointer to the printer data.
             */
            PD = (struct PrinterData *)PIO->iodrp.io_Device;
            /* And a pointer to the extended data */
            PED = (struct PrinterExtendedData *)&PD->pd_SegmentData->ps_PED;

            /* See the <devices/prtbase.h> include file for more details on
             * the PrinterData and PrinterExtendedData structures.
             */
            printf("Printername: '%s', Version: %ld, Revision: %ld\n",
            PED->ped_PrinterName, PD->pd_SegmentData->ps_Version,
            PD->pd_SegmentData->ps_Revision);

            printf("PrinterClass: 0x%lx, ColorClass: 0x%lx\n",
            PED->ped_PrinterClass, PED->ped_ColorClass);

            printf("MaxColumns: %ld, NumCharSets: %ld, NumRows: %ld\n",
                PED->ped_MaxColumns, PED->ped_NumCharSets,PED->ped_NumRows);

            printf("MaxXDots: %ld, MaxYDots: %ld, XDotsInch: %ld, YDotsInch: %ld\n",
                PED->ped_MaxXDots, PED->ped_MaxYDots, PED->ped_XDotsInch, PED->ped_YDotsInch);

            CloseDevice((struct IORequest *)PIO);
            }
         else
            printf("Can't open printer.device\n");
        DeleteExtIO((struct IORequest *)PIO);
        }
    else
        printf("Can't CreateExtIO\n");
    DeletePort((struct MsgPort *)PrinterMP);
    }
else
   printf("Can't CreatePort\n");
}
```

Reading and Changing the Printer Preferences Settings

The user preferences can be read and changed without running the Workbench Preferences tool. Reading printer preferences can be done by referring to **PD->pd_Preferences**. Listed on the next page are the printer Preferences fields and their valid ranges.

Text Preferences

PrintPitch	–	PICA, ELITE, FINE
PrintQuality	–	DRAFT, LETTER
PrintSpacing	–	SIX_LPI, EIGHT_LPI
PrintLeftMargin	–	1 to PrintRightMargin
PrintRightMargin	–	PrintLeftMargin to 999
PaperLength	–	1 to 999
PaperSize	–	US_LETTER, US_LEGAL, N_TRACTOR, W_TRACTOR, CUSTOM
PaperType	–	FANFOLD, SINGLE

Graphic Preferences

PrintImage	–	IMAGE_POSITIVE, IMAGE_NEGATIVE
PrintAspect	–	ASPECT_HORIZ, ASPECT_VERT
PrintShade	–	SHADE_BW, SHADE_GREYSCALE, SHADE_COLOR
PrintThreshold	–	1 to 15
PrintFlags	–	CORRECT_RED, CORRECT_GREEN, CORRECT_BLUE, CENTER_IMAGE, IGNORE_DIMENSIONS, BOUNDED_DIMENSIONS, ABSOLUTE_DIMENSIONS, PIXEL_DIMENSIONS, MULTIPLY_DIMENSIONS, INTEGER_SCALING, ORDERED_DITHERING, HALFTONE_DITHERING, FLOYD_DITHERING, ANTI_ALIAS, GREY_SCALE2
PrintMaxWidth	–	0 to 65535
PrintMaxHeight	–	0 to 65535
PrintDensity	–	1 to 7
PrintXOffset	–	0 to 255

This example program changes various settings in the printer device's copy of preferences.

```
/*
 * Set_Prefs.c
 *
 * This example changes the printer device's COPY of preferences (as obtained
 * when the printer device was opened by a task via OpenDevice()).  Note that
 * it only changes the printer device's copy of preferences, not the preferences
 * as set by the user via the preference editor(s).
 *
 * Compile with SAS C 5.10: LC -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>
#include <intuition/preferences.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>
```

```
struct Library *IntuitionBase;
struct Library *GfxBase;

union printerIO
{
    struct IOStdReq    ios;
    struct IODRPReq    iodrp;
    struct IOPrtCmdReq iopc;
};

struct MsgPort *PrintMP;
union printerIO *pio;

char message[] = "\
This is a test message to see how this looks when printed\n\
using various printer settings.\n\n";

VOID main(VOID);
VOID DoPrinter(VOID);
int DoTest(VOID);

VOID main(VOID)
{

if (IntuitionBase = OpenLibrary("intuition.library",0L))
    {
    if (GfxBase = OpenLibrary("graphics.library",0L))
        {
        DoPrinter();
        CloseLibrary(GfxBase);
        }

    CloseLibrary(IntuitionBase);
    }
}

VOID DoPrinter(VOID)
{

if (PrintMP = CreatePort(0L,0L))
    {
    if (pio = (union printerIO *)CreateExtIO(PrintMP,sizeof(union printerIO)))
        {
        if (!(OpenDevice("printer.device",0L,(struct IORequest *)pio,0L)))
            {
            DoTest();
            CloseDevice((struct IORequest *)pio);
            }
        DeleteExtIO((struct IORequest *)pio);
        }
    DeletePort(PrintMP);
    }
}

DoTest(VOID)
{
struct PrinterData *PD;
struct Preferences *prefs;
UWORD newpitch;
UWORD newspacing;

/* Send INIT sequence - make sure printer is inited - some      */
/* printers may eject the current page if necessary when inited */

pio->ios.io_Command = CMD_WRITE;
pio->ios.io_Data = "\033#1";
pio->ios.io_Length = -1L;

if (DoIO((struct IORequest *)pio))
    return(FALSE);

/* Print some text using the default settings from Preferences */

pio->ios.io_Command = CMD_WRITE;
pio->ios.io_Data = message;
pio->ios.io_Length = -1L;
if(DoIO((struct IORequest *)pio))
   return(FALSE);

/* Now try changing some settings
 * Note that we could just as well send the printer.device escape
 * sequences to change these settings, but this is an example program.
 */

/* Get pointer to printer data  */
PD = (struct PrinterData *) pio->ios.io_Device;

/* Get pointer to printer's copy of preferences
 * Note that the printer.device makes a copy of preferences when
 * the printer.device is successfully opened via OpenDevice(),
 * so we are only going to change the COPY of preferences
 */

prefs = &PD->pd_Preferences;


/* Change a couple of settings                              */

if (prefs->PrintSpacing == SIX_LPI)
    newspacing = EIGHT_LPI;
if (prefs->PrintSpacing == EIGHT_LPI)
    newspacing = SIX_LPI;

if (prefs->PrintPitch == PICA)
    newpitch = ELITE;
if (prefs->PrintPitch == ELITE)
    newpitch = FINE;
if (prefs->PrintPitch == FINE)
    newpitch = PICA;

/* And let's change the margins too for this example */

prefs->PrintLeftMargin = 20;
prefs->PrintRightMargin = 40;

prefs->PrintPitch = newpitch;
prefs->PrintSpacing = newspacing;

/* Send INIT sequence so that these settings are used */

pio->ios.io_Command = CMD_WRITE;
pio->ios.io_Data = "\033#1";
pio->ios.io_Length = -1L;

if(DoIO((struct IORequest *)pio))
   return(FALSE);

pio->ios.io_Command = CMD_WRITE;
pio->ios.io_Data = message;
pio->ios.io_Length = -1L;

if(DoIO((struct IORequest *)pio))
   return(FALSE);

/* Send FormFeed so page is ejected  */

pio->ios.io_Command = CMD_WRITE;
pio->ios.io_Data = "\014";
pio->ios.io_Length = -1L;

if(DoIO((struct IORequest *)pio))
   return(FALSE);

return(TRUE);
}
```

Do Your Duty. The application program is responsible for range checking if the user is able to change the preferences from within the application.

Querying the Printer Device

The status of the printer port and registers can be determined by querying the printer device. The information returned will vary depending on the type of printer—parallel or serial—selected by the user. If parallel, the data returned will reflect the current state of the parallel port; if serial, the data returned will reflect the current state of the serial port.

You query the printer device by passing an **IOStdReq** to the device with PRD_QUERY set in **io_Command** and a pointer to a structure to hold the status set in **io_Data**.

```
struct PStat
{
    UBYTE LSB;        /* least significant byte of status */
    UBYTE MSB;        /* most significant byte of status */
};

union printerIO *PrintIO;

struct PStat status;

PrintIO->ios.io_Data = &status;      /* point to status structure */
PrintIO->ios.io_Command = PRD_QUERY;
DoIO((struct IORequest *)request);
```

The status is returned in the two UBYTES set in the **io_Data** field. The printer type, either serial or parallel, is returned in the **io_Actual** field.

io_Data	**Bit**	**Active**	**Function (Serial Device)**
LSB	0	low	reserved
	1	low	reserved
	2	low	reserved
	3	low	Data Set Ready
	4	low	Clear To Send
	5	low	Carrier Detect
	6	low	Ready To Send
	7	low	Data Terminal Ready
MSB	8	high	read buffer overflow
	9	high	break sent (most recent output)
	10	high	break received (as latest input)
	11	high	transmit x-OFFed
	12	high	receive x-OFFed
	13-15	high	reserved

io_Data	**Bit**	**Active**	**Function (Parallel Device)**
LSB	0	high	printer busy (offline)
	1	high	paper out
	2	high	printer selected
	3	——	read=0; write=1
	4–7		reserved
MSB	8–15	reserved	
io_Actual			1-parallel, 2-serial

Error Codes from the Printer Device

The printer device returns error codes whenever an operation is attempted. There are two types of error codes that can be returned. Printer device error codes have positive values; Exec I/O error codes have negative values. Therefore, an application should check for a *non-zero* return code as evidence of an error, not simply a value greater than zero.

```
PrintIO->ios.io_Length   = strlen(outbuffer);
PrintIO->ios.io_Data     = (APTR)outbuffer;
PrintIO->ios.io_Command  = PRD_RAWWRITE;
if (DoIO((struct IORequest *)PrintIO))
    printf("RAW Write failed.  Error: %d ",PrintIO->ios.io_Error);
```

The error is found in **io_Error**.

Printer Device Error Codes

Error	Value	Explanation
PDERR_NOERR	0	Operation successful
PDERR_CANCEL	1	User canceled request
PDERR_NOTGRAPHICS	2	Printer cannot output graphics
PDERR_INVERTHAM	3	OBSOLETE
PDERR_BADDIMENSION	4	Print dimensions are illegal
PDERR_DIMENSIONOVERFLOW	5	OBSOLETE
PDERR_INTERNALMEMORY	6	No memory available for internal variables
PDERR_BUFFERMEMORY	7	No memory available for print buffer

Exec Error Codes

Error	Value	Explanation
IOERR_OPENFAIL	-1	Device failed to open
IOERR_ABORTED	-2	Request terminated early (after **AbortIO()**)
IOERR_NOCMD	-3	Command not supported by device
IOERR_BADLENGTH	-4	Not a valid length

Dumping a Rastport to a Printer

You dump a **RastPort** (drawing area) to a graphics capable printer by passing an **IODRPReq** to the device with PRD_DUMPRPORT set in **io_Command** along with several parameters that define how the dump is to be rendered.

```
union printerIO *PrintIO
struct RastPort *rastPort;
struct ColorMap *colorMap;
ULONG modeid;
UWORD sx, sy, sw, sh;
LONG dc, dr;
UWORD s;

PrintIO->iodrp.io_RastPort = rastPort;       /* pointer to RastPort */
PrintIO->iodrp.io_ColorMap = colorMap;       /* pointer to color map */
PrintIO->iodrp.io_Modes = modeid;            /* ModeID of ViewPort */
PrintIO->iodrp.io_SrcX = sx;                 /* RastPort X offset */
PrintIO->iodrp.io_SrcY = sy;                 /* RastPort Y offset */
```

```
PrintIO->iodrp.io_SrcWidth = sw;            /* print width from X offset */
PrintIO->iodrp.io_SrcHeight = sh;           /* print height from Y offset */
PrintIO->iodrp.io_DestCols = dc;            /* pixel width */
PrintIO->iodrp.io_DestRows = dr;            /* pixel height */
PrintIO->iodrp.io_Special = s;              /* flags */
PrintIO->iodrp.io_Command = PRD_DUMPRPORT;
SendIO((struct IORequest *)request);
```

The asynchronous **SendIO()** routine is used in this example instead of the synchronous **DoIO()** . A call to **DoIO()** does not return until the I/O request is finished. A call to **SendIO()** returns immediately. This allows your task to do other processing such as checking if the user wants to abort the I/O request. It should also be used when writing a lot of text or raw data with CMD_WRITE and PRD_RAWWRITE.

Here is an overview of the possible arguments for the **RastPort** dump.

io_RastPort	A pointer to a **RastPort**. The **RastPort**'s bitmap could be in Fast memory.
io_ColorMap	A pointer to a **ColorMap**. This could be a custom one.
io_Modes	The viewmode flags or the **ModeID** returned from **GetVPModeID()** (V36).
io_SrcX	X offset in the **RastPort** to start printing from.
io_SrcY	Y offset in the **RastPort** to start printing from.
io_SrcWidth	Width of the **RastPort** to print from **io_SrcX**.
io_SrcHeight	Height of the **RastPort** to print from **io_SrcY**.
io_DestCols	Width of the dump in printer pixels.
io_DestRows	Height of the dump in printer pixels.
io_Special	Flag bits (described below).

Looking at these arguments you can see the enormous flexibility the printer device offers for dumping a **RastPort**. The **RastPort** pointed to could be totally custom defined. This flexibility means it is possible to build a **BitMap** with the resolution of the printer. This would result in having one pixel of the **BitMap** correspond to one pixel of the printer. In other words, only the resolution of the output device would limit the final result. With 12 bit planes and a custom **ColorMap**, you could dump 4096 colors—without the HAM limitation—to a suitable printer. The offset, width and height parameters allow dumps of any desired part of the picture. Finally the **ViewPort** mode, **io_DestCols**, **io_DestRows** parameters, together with the **io_Special** flags define how the dump will appear on paper and aid in getting the correct aspect ratio.

PRINTER SPECIAL FLAGS

The printer special flags (**io_Flags**) of the **IODRPReq** provide a high degree of control over the printing of a **RastPort**.

SPECIAL_ASPECT	Allows one of the dimensions to be reduced/expanded to preserve the correct aspect ratio of the printout.
SPECIAL_CENTER	Centers the image between the left and right edge of the paper.
SPECIAL_NOFORMFEED	Prevents the page from being ejected after a graphics dump. Usually used to mix graphics and text or multiple graphics dump on a page oriented printer (normally a laser printer).

SPECIAL_NOPRINT	The print size will be computed, and set in **io_DestCols** and **io_DestRows,** but won't print. This way the application can see what the actual printsize in printerpixels would be.
SPECIAL_TRUSTME	Instructs the printer not to send a reset before and after the dump. This flag is obsolete for V1.3 (and higher) drivers.
SPECIAL_DENSITY1-7	This flag bit is set by the user in Preferences. Refer to "Reading and Changing the Printer Preferences Settings" if you want to change to density of the printout. (Or any other setting for that matter.)
SPECIAL_FULLCOLS	The width is set to the maximum possible, as determined by the printer or the configuration limits.
SPECIAL_FULLROWS	The height is set to the maximum possible, as determined by the printer or the configuration limits.
SPECIAL_FRACCOLS	Informs the printer device that the value in **io_DestCols** is to be taken as a longword binary fraction of the maximum for the dimension. For example, if **io_DestCols** is 0x8000, the width would be 1/2 (0x8000 / 0xffff) of the width of the paper.
SPECIAL_FRACROWS	Informs the printer device that the value in **io_DestRows** is to be taken as a longword binary fraction for the dimension.
SPECIAL_MILCOLS	Informs the printer device that the value in **io_DestCols** is specified in thousandths of an inch. For example, if **io_DestCols** is 8000, the width of the printout would be 8.000 inches.
SPECIAL_MILROWS	Informs the printer device that the value in **io_DestRows** is specified in thousandths of an inch.

The flags are defined in the include file *devices/printer.h.*

PRINTING WITH CORRECTED ASPECT RATIO

Using the special flags it is fairly easy to ensure a graphic dump will have the correct aspect ratio on paper. There are some considerations though when printing a non-displayed **RastPort**. One way to get a corrected aspect ratio dump is to calculate the printer's ratio from **XDotsInch** and **YDotsInch** (taking into account that the printer may not have square pixels) and then adjust the width and height parameters accordingly. You then ask for a non-aspect-ratio-corrected dump since you already corrected it yourself.

Another possibility is having the printer device do it for you. To get a correct calculation you could build your **RastPort** dimensions in two ways:

1. Using an integer multiple of one of the standard (NTSC) display resolutions and setting the **io_Modes** argument accordingly. For example if your **RastPort** dimensions were 1280 x 800 (an even multiple of 640 x 400) you would set **io_Modes** to LACE | HIRES. Setting the SPECIAL_ASPECT flag would enable the printer device to properly calculate the aspect ratio of the image.

2. When using an arbitrary sized **RastPort**, you can supply the ModeID of a display mode which has the aspect ratio you would like for your **RastPort**. The aspect ratio of the various display modes are defined as ticks-per-pixel in the **Resolution** field of the **DisplayInfo** structure. You can obtain this value from the graphics database. For example, the resolution of Productivity Mode is 22:22, in other words, 1:1, perfect for a **RastPort** sized to the limits of the output device. See the "Graphics Library" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for general information on the graphics system.

The following example will dump a **RastPort** to the printer and wait for either the printer to finish or the user to cancel the dump and act accordingly.

```
/* Demo_Dump.c
 *
 * Simple example of dumping a rastport to the printer, changing
 * printer preferences programmatically and handling error codes.
 *
 * Compile with SAS C 5.10a. lc -cfist -v -L Demo_Dump
 *
 * Requires Kickstart V37
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/ports.h>
#include <devices/printer.h>
#include <devices/prtbase.h>
#include <dos/dos.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>
#include <graphics/displayinfo.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

union printerIO
{
    struct IOStdReq    ios;
    struct IODRPReq    iodrp;
    struct IOPrtCmdReq iopc;
};

struct EasyStruct reqES =
{
    sizeof(struct EasyStruct), 0, "DemoDump",
    "%s",
    NULL,
};

/* Possible printer.device and I/O errors */
static UBYTE *ErrorText[] =
{
    "PDERR_NOERR",
    "PDERR_CANCEL",
    "PDERR_NOTGRAPHICS",
    "INVERTHAM",              /* OBSOLETE */
    "BADDIMENSION",
    "DIMENSIONOVFLOW",  /* OBSOLETE */
    "INTERNALMEMORY",
    "BUFFERMEMORY",
    /* IO_ERRs */
    "IOERR_OPENFAIL",
    "IOERR_ABORTED",
    "IOERR_NOCMD",
    "IOERR_BADLENGTH"
};
```

```
/* Requester Action text */
static UBYTE *ActionText[] =
{
    "OK|CANCEL",
    "Continue",
    "Abort",
};

#define OKCANCELTEXT 0
#define CONTINUETEXT 1
#define ABORTTEXT    2

VOID main(VOID);

VOID main(VOID)
{
struct MsgPort  *PrinterMP;
union printerIO *PIO;
struct PrinterData *PD;
struct PrinterExtendedData *PED;
struct Screen *pubscreen;
struct ViewPort *vp;
STRPTR textbuffer;
LONG modeID, i,j;
ULONG dcol[5], drow[5];
ULONG signal;

/* Fails silently if not V37 or greater. Nice thing to do would be to put up
 * a V33 requester of course.
 */

/* Set up once */
reqES.es_GadgetFormat = ActionText[CONTINUETEXT];

if (IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", 37))
  {
  /* Using graphics.library to get the displaymodeID of the public screen,
   * which we'll pass to the printer.device.
   */
  if (GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 37))
    {
    if (textbuffer = (STRPTR)AllocMem(256, MEMF_CLEAR))
      {
      /* Create non-public messageport. Since we depend on V37 already, we'll
       * use the new Exec function.
       */
      if (PrinterMP = CreateMsgPort())
        {
        /* Allocate printerIO union */
        if (PIO = (union printerIO *)CreateExtIO(PrinterMP, sizeof(union printerIO)))
          {
          /* Open the printer.device */
          if (!(OpenDevice("printer.device",0,(struct IORequest *)PIO,0)))
            {
            /* Yahoo, we've got it.
             * We'll use the PrinterData structure to get to the the printer
             * preferences later on. The PrinterExtendedData structure will
             * reflect the changes we'll make to the preferences.
             */

            PD = (struct PrinterData *)PIO->iodrp.io_Device;
            PED = (struct PrinterExtendedData *)&PD->pd_SegmentData->ps_PED;

            /* We're all set. We'll grab the default public screen (normally
             * Workbench) and see what happens when we dump it with different
             * densities.
             * Next we'll put up a nice requester for the user and ask if
             * (s)he wants to actually do the dump.
             */

            if (pubscreen = LockPubScreen(NULL))
              {
              vp = &(pubscreen->ViewPort);
              /* Use graphics.library/GetVPModeID() to get the ModeID of the screen. */
              if ((modeID = GetVPModeID(vp)) != INVALID_ID)
                {
```

```
/* Seems we got a valid ModeID for the default public screen (surprise).
 * Do some fake screen dumps with densities 1, 3, 5 and 7. Depending on
 * the driver, one or more may be the same.
 */

/* Fill in those parts of the IODRPRequest which won't change */
PIO->iodrp.io_Command = PRD_DUMPRPORT;
PIO->iodrp.io_RastPort = &(pubscreen->RastPort);
PIO->iodrp.io_ColorMap = vp->ColorMap;
PIO->iodrp.io_Modes = modeID;
PIO->iodrp.io_SrcX = pubscreen->LeftEdge;
PIO->iodrp.io_SrcY = pubscreen->TopEdge;
PIO->iodrp.io_SrcWidth = pubscreen->Width;
PIO->iodrp.io_SrcHeight = pubscreen->Height;

for (i = 1,j=0; i < 8; i+=2,j++)
  {
  /* On return these will contain the actual dump dimension */
  PIO->iodrp.io_DestCols = 0;
  PIO->iodrp.io_DestRows = 0;
  /* We'll simply change our local copy of the
   * Preferences structure. Likewise we could change
   * all printer-related preferences.
   */
  PD->pd_Preferences.PrintDensity = i;
  PIO->iodrp.io_Special = SPECIAL_NOPRINT|SPECIAL_ASPECT;

  /* No need to do asynchronous I/O here */
  DoIO((struct IORequest *)PIO);

  if (PIO->iodrp.io_Error == 0)
    {
    dcol[j] = PIO->iodrp.io_DestCols;
    drow[j] = PIO->iodrp.io_DestRows;
    }
  else
    {
    j = PIO->iodrp.io_Error;
    if (j < 0)
      j = j * -1 + 7;

    sprintf(textbuffer, "Error: %s\n", ErrorText[j]);
    reqES.es_GadgetFormat = ActionText[CONTINUETEXT];
    EasyRequest(NULL, &reqES, NULL, textbuffer);
    break;
    }
  }
/* Simple, lazy way to check if we encountered any problems */
if (i == 9)
  {
  /* Build an 'intelligent' requester */
  sprintf(textbuffer,
    "%s: %5ld x %5ld\n%s: %5ld x %5ld\n%s: %5ld x%5ld\n%s: %5ld x %5ld\n\n%s",
    "Density 1", dcol[0], drow[0],
    "Density 3", dcol[1], drow[1],
    "Density 5", dcol[2], drow[2],
    "Density 7", dcol[3], drow[3],
    "Print screen at highest density?");
  reqES.es_GadgetFormat = ActionText[OKCANCELTEXT];

  /* Obviously the choice presented to the user here is a very
   * simple one. To print or not to print. In a real life
   * application, a requester could be presented, inviting
   * the user to select density, aspect, dithering etc.
   * The fun part is, of course, that the user can, to a certain
   * degree, be informed about the effects of her/his selections.
   */
  if (EasyRequest(NULL, &reqES, NULL, textbuffer))
    {
    /* We've still got the density preference set to the highest
     * density, so no need to change that.
     * All we do here is re-initialize io_DestCols/Rows and remove
     * the SPECIAL_NOPRINT flag from io_Special.
     */
    PIO->iodrp.io_DestCols = 0;
    PIO->iodrp.io_DestRows = 0;
    PIO->iodrp.io_Special &= ~SPECIAL_NOPRINT;
```

```
                    /* Always give the user a change to abort.
                     * So we'll use SendIO(), instead of DoIO(), to be asynch and
                     * catch a possible user request to abort printing. Normally,
                     * the user would be presented with a nice, fat, ABORT requester.
                     * However, since this example doesn't even open a window, and is
                     * basically a 'GraphicDumpDefaultPubscreen' equivalent, we'll use
                     * CTRL-C as the user-abort. Besides that, got to keep it short.
                     */
                    SendIO((struct IORequest *)PIO);

                    /* Now Wait() for either a user signal (CTRL-C) or a signal from
                     * the printer.device
                     */
                    signal = Wait(1 << PrinterMP->mp_SigBit | SIGBREAKF_CTRL_C);

                    if (signal & SIGBREAKF_CTRL_C)
                      {
                      /* User wants to abort */
                      AbortIO((struct IORequest *)PIO);
                      WaitIO((struct IORequest *)PIO);
                      }

                    if (signal & (1 << PrinterMP->mp_SigBit))
                      {
                      /* printer is either ready or an error has occurred */
                      /* Remove any messages */
                      while(GetMsg(PrinterMP));
                      }
                    /* Check for errors (in this case we count user-abort as an error) */
                    if (PIO->iodrp.io_Error != 0)
                      {
                      j = PIO->iodrp.io_Error;
                      if (j < 0)
                        j = j * -1 + 7;
                      sprintf(textbuffer, "Error: %s\n", ErrorText[j]);
                      reqES.es_GadgetFormat = ActionText[CONTINUETEXT];
                      EasyRequest(NULL, &reqES, NULL, textbuffer);
                      }

                    } /* else user doesn't want to print */
                  }
                }
              else
                /* Say what? */
                EasyRequest(NULL, &reqES, NULL, "Invalid ModeID\n");
              UnlockPubScreen(NULL, pubscreen);
              }
            else
              EasyRequest(NULL, &reqES, NULL, "Can't lock Public Screen\n");

            CloseDevice((struct IORequest *)PIO);
            }
          else
            EasyRequest(NULL, &reqES, NULL, "Can't open printer.device\n");

          DeleteExtIO((struct IORequest *)PIO);
          }
        else
          EasyRequest(NULL, &reqES, NULL, "Can't create Extented I/O Request\n");
        DeleteMsgPort(PrinterMP);
        }
      else
        EasyRequest(NULL, &reqES, NULL, "Can't create Message port\n");
      /* else Out of memory? 256 BYTES? */
      FreeMem(textbuffer,256);
      }
    CloseLibrary(GfxBase);
    } /* else MAJOR confusion */
  CloseLibrary(IntuitionBase);
  }
}
```

STRIP PRINTING

Strip printing is a method which allows you to print a picture that normally requires a large print buffer when there is not much memory available. This would allow, for example, a **RastPort** to be printed at a higher resolution than it was drawn in. Strip printing is done by creating a temporary **RastPort** as wide as the source **RastPort**, but not as high. The source **RastPort** is then rendered, a strip at a time, into the temporary **RastPort** which is dumped to the printer.

The height of the strip to dump must be an integer multiple of the printer's **NumRows** if a non-aspect-ratio-corrected image is to be printed.

For an aspect-ratio-corrected image, the SPECIAL_NOPRINT flag will have to be used to find an **io_DestRows** that is an integer multiple of **NumRows**. This can be done by varying the source height and asking for a SPECIAL_NOPRINT dump until **io_DestRows** holds a number that is an integer multiple of the printer's **NumRows**.

If smoothing is to work with strip printing, a raster line above and below the actual area should be added. The line above should be the last line from the previous strip, the line below should be the first line of the next strip. Of course, the first strip should not have a line added above and the last strip should not have a line added below.

The following is a strip printing procedure for a **RastPort** which is 200 lines high.

First strip

- copy source line 0 through 50 (51 lines) to strip **RastPort** lines 0 through 50 (51 lines).
- **io_SrcY** = 0, **io_Height** = 50.
- the printer device can see there is no line above the first line to dump (since SrcY = 0) and that there is a line below the last line to dump (since there is a 51 line **RastPort** and only 50 lines are dumped).

Second strip

- copy source line 49 through 100 (52 lines) to strip **RastPort** lines 0 through 51 (52 lines).
- **io_SrcY** = 1, **io_Height** = 50.
- the printer device can see there is a line above the first line to dump (since SrcY = 1) and that there is a line below the last line to dump (since there is a 52 line **RastPort** and only 50 lines are dumped).

Third strip

- copy source line 99 through 150 (52 lines) to strip **RastPort** lines 0 through 51 (52 lines).
- **io_SrcY** = 1, **io_Height** = 50.
- the printer device can see there is a line above the first line to dump (since SrcY = 1) and that there is a line below the last line to dump (since there is a 52 line **RastPort** and only 50 lines are dumped).

Fourth strip

- copy source line 149 through 199 (51 lines) to strip **RastPort** lines 0 through 50 (51 lines).
- **io_SrcY** = 1, **io_Height** = 50.
- the printer device can see there is a line above the first line to dump (since SrcY = 1) and that there is no line below the last line to dump (since there is a 51 line **RastPort** and only 50 lines are dumped).

ADDITIONAL NOTES ABOUT GRAPHIC DUMPS

1. When dumping a 1 bitplane image select the black and white mode in Preferences. This is much faster than a grey-scale or color dump.
2. Horizontal dumps are much faster than vertical dumps.
3. Smoothing doubles the print time. Use it for final copy only.
4. F-S dithering doubles the print time. Ordered and half-tone dithering incur no extra overhead.
5. The lower the density, the faster the printout.
6. Friction-fed paper tends to be much more accurate than tractor-fed paper in terms of vertical dot placement (i.e., less horizontal strips or white lines).
7. Densities which use more than one pass tend to produce muddy grey-scale or color printouts. It is recommended not to choose these densities when doing a grey-scale or color dump.

   Keep This in Mind. It is possible that the printer has been instructed to receive a certain amount of data and is still in an "expecting" state if an I/O request has been aborted by the user. This means the printer would try to finish the job with the data the next I/O request might send. Currently the best way to overcome this problem is for the printer to be reset.

Creating a Printer Driver

Creating the printer-dependent modules for the printer device involves writing the data structures and code, compiling and assembling them, and linking to produce an Amiga binary object file. The modules a driver contains varies depending on whether the printer is non-graphics or graphics capable.

All drivers contain these modules:

macros.i	–	include file for init.asm, contains printer device macro definitions
printertag.asm	–	printer specific capabilities such as density, character sets and color
init.asm	–	opens the various libraries required by the printer driver. This will be the same for all printers
data.c	–	contains printer device RAW commands and the extended character set supported by the printer
dospecial.c	–	printer specific special processing required for printer device commands like aSLRM and aSFC

Graphic printer drivers require these additional modules:

render.c	–	printer specific processing to do graphics output and fill the output buffer
transfer.c	–	printer specific processing called by render.c to output the buffer to the printer. Code it in assembly if speed is important
density.c	–	printer specific processing to construct the proper print density commands

The first piece of the printer driver is the **PrinterSegment** structure described in *devices/prtbase.h* (this is pointed to by the BPTR returned by the **LoadSeg()** of the object file). The **PrinterSegment** contains the **PrinterExtendedData** (PED) structures (also described in *devices/prtbase.h*) at the beginning of the object. The PED structure contains data describing the capabilities of the printer, as well as pointers to code and other data. Here is the assembly code for a sample **PrinterSegment**, which would be linked to the beginning of the sequence of files as *printertag.asm*.

```
**********************************************************************
*
*       printer device dependent code tag
*
*
*
**********************************************************************

        SECTION         printer

*------ Included Files -----------------------------------------------

        INCLUDE         "exec/types.i"
        INCLUDE         "exec/nodes.i"
        INCLUDE         "exec/strings.i"

        INCLUDE         "epsonX_rev.i"  ; contains VERSION & REVISION

        INCLUDE         "devices/prtbase.i"


*------ Imported Names -----------------------------------------------

        XREF            _Init
        XREF            _Expunge
        XREF            _Open
        XREF            _Close
        XREF            _CommandTable
        XREF            _PrinterSegmentData
        XREF            _DoSpecial
        XREF            _Render
        XREF            _ExtendedCharTable

*------ Exported Names ----------------------------------------------

        XDEF            _PEDData

**********************************************************************
        ; in case anyone tries to execute this
        MOVEQ   #0,D0
        RTS

        DC.W    VERSION         ; must be at least 35 (V1.3 and up)
        DC.W    REVISION        ; your own revision number
_PEDData:
        DC.L    printerName     ; pointer to the printer name
        DC.L    _Init           ; pointer to the initialization code
        DC.L    _Expunge        ; pointer to the expunge code
        DC.L    _Open           ; pointer to the open code
        DC.L    _Close          ; pointer to the close code
        DC.B    PPC_COLORGFX    ; PrinterClass
        DC.B    PCC_YMCB        ; ColorClass
        DC.B    136             ; MaxColumns
        DC.B    10              ; NumCharSets
        DC.W    8               ; NumRows
        DC.L    1632            ; MaxXDots
        DC.L    0               ; MaxYDots
        DC.W    120             ; XDotsInch
        DC.W    72              ; YDotsInch
        DC.L    _CommandTable   ; pointer to Command strings
        DC.L    _DoSpecial      ; pointer to Command Code function
        DC.L    _Render         ; pointer to Graphics Render function
        DC.L    30              ; Timeout

        DC.L    _ExtendedCharTable  ; pointer to 8BitChar table

        DS.L    1               ; Flag for PrintMode (reserve space)
        DC.L    0               ; pointer to ConvFunc (char conversion function)

printerName:
        DC.B    'EpsonX'
        DC.B    0
        END
```

The printer name should be the brand name of the printer that is available for use by programs wishing to be specific about the printer name in any diagnostic or instruction messages. The four functions at the top of the structure are used to initialize this printer-dependent code:

(*(PED->ped_Init))(PD) ;

This is called when the printer-dependent code is loaded and provides a pointer to the printer device for use by the printer-dependent code. It can also be used to open up any libraries or devices needed by the printer-dependent code.

(*(PED->ped_Expunge))() ;

This is called immediately before the printer-dependent code is unloaded, to allow it to close any resources obtained at initialization time.

(*(PED->ped_Open))(ior) ;

This is called in the process of an **OpenDevice()** call, after the Preferences are read and the correct primitive I/O device (parallel or serial) is opened. It must return zero if the open is successful, or non-zero to terminate the open and return an error to the user.

(*(PED->ped_Close))(ior) ;

This is called in the process of a **CloseDevice()** call to allow the printer-dependent code to close any resources obtained at open time.

The **pd_** variable provided as a parameter to the initialization call is a pointer to the **PrinterData** structure described in *devices/prtbase.h*. This is also the same as the **io_Device** entry in printer I/O requests.

pd_SegmentData
: This points back to the **PrinterSegment**, which contains the PED.

pd_PrintBuf
: This is available for use by the printer-dependent code—it is not otherwise used by the printer device.

(*pd_PWrite)(data, length);
: This is the interface routine to the primitive I/O device. This routine uses two I/O requests to the primitive device, so writes are double-buffered. The data parameter points to the byte data to send, and the length is the number of bytes.

(*pd_PBothReady)();
: This waits for both primitive I/O requests to complete. This is useful if your code does not want to use double buffering. If you want to use the same data buffer for successive **pd_PWrites**, you must separate them with a call to this routine.

pd_Preferences
: This is the copy of Preferences in use by the printer device, obtained when the printer was opened.

The timeout field is the number of seconds that an I/O request from the printer device to the primitive I/O device (parallel or serial) will remain posted and unsatisfied before the timeout requester is presented to the user. The timeout value should be long enough to avoid the requester during normal printing.

The **PrintMode** field is a flag which indicates whether text has been printed or not (1 means printed, 0 means not printed). This flag is used in drivers for page oriented printers to indicate that there is no alphanumeric data waiting for a formfeed.

WRITING AN ALPHANUMERIC PRINTER DRIVER

The alphanumeric portion of the printer driver is designed to convert ANSI x3.64 style commands into the specific escape codes required by each individual printer. For example, the ANSI code for underline-on is ESC[4m. The Commodore MPS-1250 printer would like a ESC[-1 to set underline-on. The HP LaserJet accepts ESC[&dD as a start underline command. By using the printer driver, all printers may be handled in a similar manner.

There are two parts to the alphanumeric portion of the printer driver: the **CommandTable** data table and the **DoSpecial()** routine.

Command Table

The **CommandTable** is used to convert all escape codes that can be handled by simple substitution. It has one entry per ANSI command supported by the printer driver. When you are creating a custom **CommandTable**, you must maintain the order of the commands in the same sequence as that shown in *devices/printer.h*. By placing the specific codes for your printer in the proper positions, the conversion takes place automatically.

> *Octal knows NULL.* If the code for your printer requires a decimal 0 (an ASCII NULL character), you enter this NULL into the **CommandTable** as octal 376 (decimal 254).

Placing an octal value of 377 (255 decimal) in a position in the command table indicates to the printer device that no simple conversion is available on this printer for this ANSI command. For example, if a daisy-wheel printer does not have a foreign character set, 377 octal (255 decimal) is placed in that position in the command table. However, 377 in a position can also mean that the ANSI command is to be handled by code located in the **DoSpecial()** function. For future compatibility *all* printer commands should be present in the command table, and those not supported by the printer filled with the dummy entry 377 octal.

DoSpecial()

The **DoSpecial()** function is meant to implement all the ANSI functions that cannot be done by simple substitution, but can be handled by a more complex sequence of control characters sent to the printer. These are functions that need parameter conversion, read values from Preferences, and so on. Complete routines can also be placed in *dospecial.c*. For instance, in a driver for a page oriented-printer such as the HP LaserJet, the dummy **Close()** routine from the *init.asm* file would be replaced by a real **Close()** routine in *dospecial.c*. This close routine would handle ejecting the paper after text has been sent to the printer and the printer has been closed.

The **DoSpecial()** function is set up as follows:

```
#include "exec/types.h"
#include "devices/printer.h"
#include "devices/prtbase.h"

extern struct PrinterData *PD;

DoSpecial(command,outputBuffer,vline,currentVMI,crlfFlag,Parms)
UBYTE outputBuffer[];
UWORD *command;
BYTE *vline;
BYTE *currentVMI;
BYTE *crlfFlag;
UBYTE Parms[];
{               /* code begins here... */
```

where

command
: points to the command number. The *devices/printer.h* file contains the definitions for the routines to use (aRIN is initialize, and so on).

vline
: points to the value for the current line position.

currentVMI
: points to the value for the current line spacing.

crlfFlag
points to the setting of the "add line feed after carriage return" flag.

Parms
contain whatever parameters were given with the ANSI command.

outputBuffer
points to the memory buffer into which the converted command is returned.

Almost every printer will require an aRIN (initialize) command in **DoSpecial()**. This command reads the printer settings from Preferences and creates the proper control sequence for the specific printer. It also returns the character set to normal (not italicized, not bold, and so on). Other functions depend on the printer.

Certain functions are implemented both in the **CommandTable** and in the **DoSpecial()** routine. These are functions such as superscript, subscript, PLU (partial line up), and PLD (partial line down), which can often be handled by a simple conversion. However, some of these functions must also adjust the printer device's line-position variable.

> *Save the Data!* Some printers lose data when sent their own reset command. For this reason, it is recommended that if the printer's own reset command is going to be used, **PD->pd_PWaitEnabled** should be defined to be a character that the printer will not print. This character should be put in the reset string before and after the reset character(s) in the command table.

In the EpsonX[CBM_MPS-1250] **DoSpecial()** function you'll see

```
if (*command == aRIS)
    {        /* reset command */
    PD->pd_PWaitEnabled = \375; /* preserve that data! */
}
```

while in the command table the string for reset is defined as "\375\033@\375". This means that when the printer device outputs the reset string "\033@", it will first see the "\375", wait a second and output the reset string. While the printer is resetting, the printer device gets the second "\375" and waits another second. This ensures that no data will be lost if a reset command is embedded in a string.

Printertag.asm

For an alphanumeric printer the printer-specific values that need to be filled in *printertag.asm* are as follows:

MaxColumns
the maximum number of columns the printer can print across the page.

NumCharSets
the number of character sets which can be selected.

8BitChars
a pointer to an extended character table. If the field is null, the default table will be used.

ConvFunc
a pointer to a character conversion routine. If the field is null, no conversion routine will be used.

Extended Character Table

The **8BitChars** field could contain a pointer to a table of characters for the ASCII codes $A0 to $FF. The symbols for these codes are shown in the "IFF Appendix" of this manual. If this field contains a NULL, it means no specific table is provided for the driver, and the default table is to be used instead.

Care should be taken when generating this table because of the way the table is parsed by the printer device. Valid expressions in the table include \011 where 011 is an octal number, \\000 for null and \\n where n is a 1 to 3 digit decimal number. To enter an actual backslash in the table requires the somewhat awkward \\\\. As an example, here is a list of the first entries of the EpsonxX[CBM_MPS-1250] table:

```
char *ExtendedCharTable[] =
     {
     " ",                             /* NBSP */
     "\033R\007[\033R\\0",            /* i */
     "c\010|",                        /* c| */
     "\033R\003#\033R\\0",            /* L- */
     "\033R\005$\033R\\0",            /* o */
     "\033R\010\\\\\033R\\0",         /* Y- */
     "|",                             /* | */
     "\033R\002@\033R\\0",            /* SS */
     "\033R\001~\033R\\0",            /* " */
     "c",                             /* copyright */
     "\033S\\0a\010_\033T",           /* a_ */
     "<",                              /* <̄< */
     "~",                             /* - */
     "-",                             /* SHY */
     "r",                             /* registered trademark */
     "-",                             /* - */
     /* more entries go here */
};
```

Character Conversion Routine

The **ConvFunc** field contains a pointer to a character conversion function that allows you to selectively translate any character to a combination of other characters. If no translation conversion is necessary (for most printers it isn't), the field should contain a null.

ConvFunc() arguments are a pointer to a buffer, the character currently processed, and a CR/LF flag. The **ConvFunc()** function should return a -1 if no conversion has been done. If the character is not to be added to the buffer, a 0 can be returned. If any translation is done, the number of characters added to the buffer must be returned.

Besides simple character translation, the **ConvFunc()** function can be used to add features like underlining to a printer which doesn't support them automatically. A global flag could be introduced that could be set or cleared by the **DoSpecial()** function. Depending on the status of the flag the **ConvFunc()** routine could, for example, put the character, a backspace and an underline character in the buffer and return 3, the number of characters added to the buffer.

The **ConvFunc()** function for this could look like the following example:

```
#define DO_UNDERLINE    0x01
#define DO_BOLD         0x02
/* etc */

external short myflags;

int ConvFunc(buffer, c, crlf_flag)
char *buffer, c;
int crlf_flag
```

```
{
int nr_of_chars_added = 0;

/* for this example we only do this for chars in the 0x20-0x7e range */
/* Conversion of ESC (0x1b) and CSI (0x9b) is NOT recommended */

if (c > 0x1f && c < 0x7f)
    {           /* within space - ~ range ? */
    if (myflags & DO_UNDERLINE)
        {
        *buffer++ = c;              /* the character itself */
        *buffer++ = 0x08;           /* a backspace */
        *buffer++ = '_';            /* an underline char */
        nr_of_chars_added = 3;      /* added three chars to buffer */
        }
    if (myflags & DO_BOLD)
        {
        if (nr_of_chars_added)
            {     /* already have added something */
            *buffer++ = 0x08;       /* so we start with backspace */
            ++nr_of_chars_added;    /* and increment the counter */
            }
        *buffer++ = c;
        *buffer++ = 0x08;
        *buffer++ = c;
        ++nr_of_chars_added;
        if (myflags & DO_UNDERLINE)
            {     /* did we do underline too? */
            *buffer++ = 0x08;       /* then backspace again */
            *buffer++ = '_';        /* (printer goes crazy by now) */
            nr_of_chars_added += 2; /* two more chars */
            }
        }
    }
if (nr_of_chars_added)
    return(nr_of_chars_added);      /* total nr of chars we added */
else
    return(-1);                     /* we didn't do anything */
}
```

In **DoSpecial()** the flagbits could be set or cleared, with code like the following:

```
if (*command == aRIS)           /* reset  command */
    myflags = 0;                /* clear all flags */

if (*command == aRIN)           /* initialize command */
    myflags = 0;

if (*command == aSGR0)          /* 'PLAIN' command */
    myflags = 0;

if (*command == aSGR4)          /* underline on */
    myflags |= DO_UNDERLINE;    /* set underline bit */

if (*command == aSGR24)         /* underline off */
    myflags &= ~DO_UNDERLINE;   /* clear underline bit */

if (*command == aSGR1)          /* bold on */
    myflags |= DO_BOLD;         /* set bold bit */

if (*command == aSGR22)         /* bold off */
    myflags &= ~DO_BOLD;        /* clear bold bit */
```

Try to keep the expansions to a minimum so that the throughput will not be slowed down too much, and to reduce the possibility of data overrunning the printer device buffer.

WRITING A GRAPHICS PRINTER DRIVER

Designing the graphics portion of a custom printer driver consists of two steps: writing the printer-specific **Render()**, **Transfer()** and **SetDensity()** functions, and replacing the printer-specific values in *printertag.asm*. **Render()**, **Transfer()** and **SetDensity()** comprise *render.c*, *transfer.c*, and *density.c* modules, respectively.

A printer that does *not* support graphics has a very simple form of **Render()**; it returns an error. Here is sample code for **Render()** for a non-graphics printer (such as an Alphacom or Diablo 630):

```
#include "exec/types.h"
#include "devices/printer.h"
int Render()
{
    return(PDERR_NOTGRAPHICS);
}
```

The following section describes the contents of a typical driver for a printer that does support graphics.

Render()

This function is the main printer-specific code module and consists of seven parts referred to here as cases:

- Pre-Master initialization (Case 5)
- Master initialization (Case 0)
- Putting the pixels in a buffer (Case 1)
- Dumping a pixel buffer to the printer (Case 2)
- Closing down (Case 4)
- Clearing and initializing the pixel buffer (Case 3)
- Switching to the next color(Case 6) (special case for multi-color printers)

State Your Case. The numbering of the cases reflects the value of each step as a case in a C-language switch statement. It does not denote the order that the functions are executed; the order in which they are listed above denotes that.

For each case, **Render()** receives four long variables as parameters: `ct`, `x`, `y` and `status`. These parameters are described below for each of the seven cases that **Render()** must handle.

Pre-Master initialization (Case 5)

Parameters:

ct — 0 or pointer to the **IODRPReq** structure passed to **PCDumpRPort**
x — **io_Special** flag from the **IODRPReq** structure
y — 0

When the printer device is first opened, **Render()** is called with ct set to 0, to give the driver a chance to set up the density values before the actual graphic dump is called.

The parameter passed in x will be the **io_Special** flag which contains the density and other SPECIAL flags. The only flags used at this point are the DENSITY flags, all others should be ignored. Never call **PWrite()** during this case. When you are finished handling this case, return PDERR_NOERR.

Master initialization (Case 0).

Parameters:

> ct — pointer to a **IODRPReq** structure
> x — width (in pixels) of printed picture
> y — height (in pixels) of printed picture

Everything is A-OK. At this point the printer device has already checked that the values are within range for the printer. This is done by checking values listed in *printertag.asm.*

The x and y value should be used to allocate enough memory for a command and data buffer for the printer. If the allocation fails, PDERR_BUFFERMEMORY should be returned. In general, the buffer needs to be large enough for the commands and data required for one pass of the print head. These typically take the following form:

    <start gfx cmd> <data> <end gfx cmd>

The <start gfx cmd> should contain any special, one-time initializations that the printer might require such as:

- Carriage Return—some printers start printing graphics without returning the printhead. Sending a CR assures that printing will start from the left edge.
- Unidirectional—some printers which have a bidirectional mode produce non-matching vertical lines during a graphics dump, giving a wavy result. To prevent this, your driver should set the printer to unidirectional mode.
- Clear margins—some printers force graphic dumps to be done within the text margins, thus they should be cleared.
- Other commands—enter the graphics mode, set density, etc.

Multi-Pass? Don't Forget the Memory. In addition to the memory for commands and data, a multi-pass color printer must allocate enough buffer space for each of the different color passes.

The printer should never be reset during the master initialization case This will cause problems during multiple dumps. Also, the pointer to the **IODRPReq** structure in ct should not be used except for those rare printers which require it to do the dump themselves. Return the PDERR_TOOKCONTROL error in that case so that the printer device can exit gracefully.

PDERR_TOOKCONTROL, An Error in Name Only. The printer device error code, PDERR_TOOKCONTROL, is not an error at all, but an internal indicator that the printer driver is doing the graphic dump entirely on its own. The printer device can assume the dump has been done. The calling application will not be informed of this, but will receive PDERR_NOERR instead.

The example *render.c* functions listed at the end of this chapter use double buffering to reduce the dump time which is why the **AllocMem()** calls are for BUFSIZE *times two*, where BUFSIZE represents the amount of memory for one entire print cycle. However, contrary to the example source code, allocating the two buffers independently of each other is recommended. A request for one large block of contiguous memory might be refused. Two smaller requests are more likely to be granted.

Putting the pixels in a buffer (Case 1).

Parameters:

ct — pointer to a PrtInfo structure.
x — PCM color code (if the printer is PCC_MULTI_PASS).
y — printer row # (the range is 0 to pixel height - 1).

In this case, you are passed an entire row of YMCB intensity values (Yellow, Magenta, Cyan, Black). To handle this case, you call the **Transfer()** function in the *transfer.c* module. You should return PDERR_NOERR after handling this case. The PCM-defines for the x parameter from the file *devices/prtgfx.h* are PCMYELLOW, PCMMAGENTA, PCMCYAN and PCMBLACK.

Dumping a pixel buffer to the printer (Case 2).

Parameters:

ct — 0
x — 0
y — # of rows sent (the range is 1 to NumRows).

At this point the data can be Run Length Encoded (RLE) if your printer supports it. If the printer doesn't support RLE, the data should be white-space stripped. This involves scanning the buffer from end to beginning for the position of the first occurrence of a non-zero value. Only the data from the beginning of the buffer to this position should be sent to the printer. This will significantly reduce print times.

The value of y can be used to advance the paper the appropriate number of pixel lines if your printer supports that feature. This helps prevent white lines from appearing between graphic dumps.

You can also do post-processing on the buffer at this point. For example, if your printer uses the hexadecimal number $03 as a command and requires the sequence $03 $03 to send $03 as data, you would probably want to scan the buffer and expand any $03s to $03 $03 during this case. Of course, you'll need to allocate space somewhere in order to expand the buffer.

The error from **PWrite()** should be returned after this call.

Clearing and initializing the pixel buffer (Case 3)

Parameters:

ct — 0
x — 0
y — 0

The printer driver does not send blank pixels so you must initialize the buffer to the value your printer uses for blank pixels (usually 0). Clearing the buffer should be the same for all printers. Initializing the buffer is printer specific, and it includes placing the printer-specific control codes in the buffer before and after the data.

This call is made before each Case 2 call. Clear your active print buffer — remember you are double buffering—and initialize it if necessary. After this call, PDERR_NOERR should be returned.

Closing Down (Case 4).

Parameters: ct — error code
x — **io_Special** flag from the **IODRPReq** structure
y — 0

This call is made at the end of the graphic dump or if the graphic dump was cancelled for some reason. At this point you should free the printer buffer memory. You can determine if memory was allocated by checking that the value of **PD->pd_PrintBuf** is not NULL. If memory was allocated, you must wait for the print buffers to clear (by calling **PBothReady**) and then deallocate the memory. If the printer—usually a page oriented printer—requires a page eject command, it can be given here. Before you do, though, you should check the SPECIAL_NOFORMFEED bit in x. Don't issue the command if it is set.

If the error condition in *ct* is PDERR_CANCEL, you should not **PWrite()**. This error indicates that the user is trying to cancel the dump for whatever reason. Each additional **PWrite()** will generate another printer trouble requester. Obviously, this is not desirable.

During this render case **PWrite()** could be used to:

- reset the line spacing. If the printer doesn't have an advance 'n' dots command, then you'll probably advance the paper by changing the line spacing. If you do, set it back to either 6 or 8 lpi (depending on Preferences) when you are finished printing.
- set bidirectional mode if you selected unidirectional mode in render Case 0.
- set black text; some printers print the text in the last color used, even if it was in graphics mode.
- restore the margins if you cancelled the margins in render Case 0.
- any other command needed to exit the graphics mode, eject the page, etc.

Either PDERR_NOERR or the error from **PWrite()** should be returned after this call.

Switching to the next color (Case 6)

This call provides support for printers which require that colors be sent in separate passes. When this call is made, you should instruct the printer to advance its color panel. This case is only needed for printers of the type PCC_MULTI_PASS, such as the CalComp ColorMaster.

Transfer()

Transfer() dithers and renders an entire row of pixels passed to it by the **Render()** function. When **Transfer()** gets called, it is passed 5 parameters

Parameters: PInfo — a pointer to a **PrtInfo** structure
y — the row number
ptr — a pointer to the buffer
colors — a pointer to the color buffers
BufOffset — the buffer offset for interleaved printing.

The dithering process of **Transfer()** might entail thresholding, grey-scale dithering, or color-dithering each destination pixel.

If **PInfo->pi_threshold** is non-zero, then the dither value is:

```
PInfo->pi_threshold ^15.
```

If **PInfo->pi_threshold** is zero, then the dither value is computed by:

```
*(PInfo->pi_dmatrix + ((y & 3) * 2) + (x & 3))
```

where x is initialized to **PInfo->pi_xpos** and is incremented for each of the destination pixels. Since the printer device uses a 4x4 dither matrix, you must calculate the dither value exactly as given above. Otherwise, your driver will be non-standard and the results will be unpredictable.

The **Transfer()** function renders by putting a pixel in the print buffer based on the dither value. If the intensity value for the pixel is greater than the dither value as computed above, then the pixel should be put in the print buffer. If it is less than, or equal to the dither value, it should be skipped to process the next pixel.

Printer ColorClass	**Type of Dithering**	**Rendering logic**
PCC_BW	Thresholding	Test the black value against the threshold value to see if you should render a black pixel.
	Grey Scale	Test the black value against the dither value to see if you should render a black pixel.
	Color	NA
PCC_YMC	Thresholding	Test the black value against the threshold value to see if you should render a black pixel. Print yellow, magenta and cyan pixel to make black.
	Grey Scale	Test the black value against the dither value to see if you should render a black pixel. Print yellow, magenta and cyan pixel to make black.
	Color	Test the yellow value against the dither value to see if you should render a yellow pixel. Repeat this process for magenta and cyan.
PCC_YMCB	Thresholding	Test the black value against the threshold value to see if you should render a black pixel.
	Grey Scale	Test the black value against the dither value to see if you should render a black pixel.
	Color	Test the black value against the dither value to see if you should render a black pixel. If black is not rendered, then test the yellow value against the dither value to see if you should render a yellow pixel. Repeat this process for magenta and cyan. (See the EpsonX *transfer.c* file)
PCC_YMC_BW	Thresholding	Test the black value against the threshold value to see if you should render a black pixel.
	Grey Scale	Test the black value against the dither value to see if you should render a black pixel.
	Color	Test the yellow value against the dither value to see if you should render a yellow pixel. Repeat this process for magenta and cyan.

In general, if black is rendered for a specific printer dot, then the YMC values should be ignored, since the combination of YMC is black. It is recommended that the printer buffer be constructed so that the order of colors printed is yellow, magenta, cyan and black, to prevent smudging and minimize color contamination on ribbon color printers.

The example *transfer.c* files are provided in C for demonstration only. Writing this module in assembler can cut the time needed for a graphic dump in half. The EpsonX *transfer.asm* file is an example of this.

SetDensity()

SetDensity() is a short function which implements multiple densities. It is called in the Pre-master initialization case of the **Render()** function. It is passed the density code in **density_code**. This is used to select the desired density or, if the user asked for a higher density than is supported, the maximum density available. **SetDensity()** should also handle narrow and wide tractor paper sizes.

Densities below 80 dpi should not be supported in **SetDensity()**, so that a minimum of 640 dots across for a standard 8.5x11-inch paper is guaranteed. This results in a 1:1 correspondence of dots on the printer to dots on the screen in standard screen sizes. The HP LaserJet is an exception to this rule. Its minimum density is 75 dpi because the original HP LaserJet had too little memory to output a full page at a higher density.

Printertag.asm

For a graphic printer the printer-specific values that need to be filled in in *printertag.asm* are as follows:

MaxXDots
The maximum number of dots the printer can print across the page.

MaxYDots
The maximum number of dots the printer can print down the page. Generally, if the printer supports roll or form feed paper, this value should be 0 indicating that there is no limit. If the printer has a definite y dots maximum (as a laser printer does), this number should be entered here.

XDotsInch
The dot density in x (supplied by **SetDensity()**, if it exists).

YDotsInch
The dot density in y (supplied by **SetDensity()**, if it exists).

PrinterClass
The printer class of the printer.

PPC_BWALPHA	black&white alphanumeric, no graphics.
PPC_BWGFX	black&white (only) graphics.
PPC_COLORALPHA	color alphanumeric, no graphics.
PPC_COLORGFX	color (and maybe black&white) graphics.

ColorClass
The color class the printer falls into.

PCC_BW	Black&White only
PCC_YMC	Yellow Magenta Cyan only.

PCC_YMC_BW	Yellow Magenta Cyan or Black&White, but not both
PCC_YMCB	Yellow Magenta Cyan Black
PCC_WB	White&Black only, 0 is BLACK
PCC_BGR	Blue Green Red
PCC_BGR_WB	Blue Green Red or Black&White
PCC_BGRW	Blue Green Red White

NumRows

The number of pixel rows printed by one pass of the print head. This number is used by the non-printer-specific code to determine when to make a render Case 2 call to you. You have to keep this number in mind when determining how big a buffer you'll need to store one print cycle's worth of data.

TESTING THE PRINTER DRIVER

A printer driver should be thoroughly tested before it is released. Though labor intensive, the alphanumeric part of a driver can be easily tested. The graphics part is more difficult. Following are some recommendations on how to test this part.

Start with a black and white (threshold 8), grey scale and color dump of the same picture. The color dump should be in color, of course. The grey scale dump should be like the color dump, except it will consist of patterns of black dots. The black and white dump will have solid black and solid white areas.

Next, do a dump with the **DestX** and **DestY** dots set to an even multiple of the **XDotsInch** and **YDotsInch** for the printer. For example, if the printer has a resolution of 120 x 144 dpi, a 480 x 432 dump could be done. This should produce a printed picture which covers 4 x 3 inches on paper. If the width of the picture is off, then the wrong value for **XDotsInch** has been put in *printertag.asm.* If the height of the picture is off, the wrong value for **YDotsInch** is in *printertag.asm.*

Do a color dump as wide as the printer can handle with the density set to 7.

Make sure that the printer doesn't force graphic dumps to be done within the text margins. This can easily be tested by setting the text margins to 30 and 50, the pitch to 10 cpi and then doing a graphic dump wider than 2 inches. The dump should be left justified and as wide as you instructed. If the dump starts at character position 30 and is cut off at position 50, the driver will have to be changed. These changes involve clearing the margins before the dump (Case 0) and restoring the margins after the dump (Case 4). An example of this is present in render.c source example listed at the end of this chapter.

The Invisible Setup. Before the graphic dump, some text must be sent to the printer to have the printer device initialize the printer. The "text" sent does not have to contain any printable characters (i.e., you can send a carriage return or other control characters).

As a final test, construct an image with a white background that has objects in it surrounded by white space. Dump this as black and white, grey scale and color. This will test the white-space stripping or RLE, and the ability of the driver to handle null lines. The white data areas should be separated by at least as many lines of white space as the **NumRows** value in the *printertag.asm* file.

Example Printer Driver Source Code

As an aid in writing printer drivers, source code for two different classes of printers is supplied. Both drivers have been successfully generated with Lattice C 5.10 and Lattice Assembler 5.10. The example drivers are:

EpsonX	A YMCB, 8 pin, multi-density interleaved printer.
HP_Laserjet	A black&white, multi-density, page-oriented printer.

All printer drivers use the following include file *macros.i* for init.asm.

```
**********************************************************************
*
*       printer device macro definitions
*
**********************************************************************

*------ external definition macros ----------------------------------

XREF_EXE        MACRO
        XREF            _LVO\1
                ENDM

XREF_DOS        MACRO
        XREF            _LVO\1
                ENDM

XREF_GFX        MACRO
        XREF            _LVO\1
                ENDM

XREF_ITU        MACRO
        XREF            _LVO\1
                ENDM

*------ library dispatch macros -------------------------------------

CALLEXE         MACRO
                CALLLIB _LVO\1
                ENDM

LINKEXE         MACRO
                LINKLIB _LVO\1,_SysBase
                ENDM

LINKDOS         MACRO
                LINKLIB _LVO\1,_DOSBase
                ENDM

LINKGFX         MACRO
                LINKLIB _LVO\1,_GfxBase
                ENDM

LINKITU         MACRO
                LINKLIB _LVO\1,_IntuitionBase
                ENDM
```

EPSONX

For the EpsonX driver, both the assembly and C version of Transfer() are supplied. In the Makefile the (faster) assembly version is used to generate the driver.

The EpsonX driver can be generated with the following Makefile.

```
LC = lc:lc
ASM = lc:asm
CFLAGS = -iINCLUDE: -b0 -d0 -v
ASMFLAGS = -iINCLUDE:
```

```
LINK = lc:blink
LIB = lib:lc.lib+lib:amiga.lib
OBJ = printertag.o+init.o+data.o+dospecial.o+render.o+transfer.o+density.o
TARGET = EpsonX

        @$(LC) $(CFLAGS) $*

$(TARGET): printertag.o init.o data.o dospecial.o render.o density.o transfer.o
        @$(LINK) <WITH <
        FROM $(OBJ)
        TO $(TARGET)
        LIBRARY $(LIB)
        NODEBUG SC SD VERBOSE MAP $(TARGET).map H
        <

init.o: init.asm
        @$(ASM) $(ASMFLAGS) init.asm

printertag.o: printertag.asm epsonX_rev.i
        @$(ASM) $(ASMFLAGS) printertag.asm

transfer.o: transfer.asm
        @$(ASM) $(ASMFLAGS) transfer.asm

dospecial.o: dospecial.c

data.o: data.c

density.o: density.c

render.o: render.c
```

Epsonx: macros.I

```
**********************************************************************
*
*       printer device macro definitions
*
**********************************************************************

*------ external definition macros ----------------------------------

XREF_EXE        MACRO
        XREF            _LVO\1
                ENDM

XREF_DOS        MACRO
        XREF            _LVO\1
                ENDM

XREF_GFX        MACRO
        XREF            _LVO\1
                ENDM

XREF_ITU        MACRO
        XREF            _LVO\1
                ENDM

*------ library dispatch macros -------------------------------------

CALLEXE         MACRO
                CALLLIB _LVO\1
                ENDM

LINKEXE         MACRO
                LINKLIB _LVO\1,_SysBase
                ENDM

LINKDOS         MACRO
                LINKLIB _LVO\1,_DOSBase
                ENDM

LINKGFX         MACRO
                LINKLIB _LVO\1,_GfxBase
                ENDM
```

```
LINKITU         MACRO
                LINKLIB _LVO\1,_IntuitionBase
                ENDM
```

Epsonx: printertag.asm

```
**********************************************************************
*
*       printer device dependent code tag
*
**********************************************************************

        SECTION         printer

*------ Included Files -----------------------------------------------

        INCLUDE         "exec/types.i"
        INCLUDE         "exec/nodes.i"
        INCLUDE         "exec/strings.i"

        INCLUDE         "epsonX_rev.i"

        INCLUDE         "devices/prtbase.i"

*------ Imported Names -----------------------------------------------

        XREF            _Init
        XREF            _Expunge
        XREF            _Open
        XREF            _Close
        XREF            _CommandTable
        XREF            _PrinterSegmentData
        XREF            _DoSpecial
        XREF            _Render
        XREF            _ExtendedCharTable

*------ Exported Names -----------------------------------------------

        XDEF            _PEDData

**********************************************************************

                MOVEQ   #0,D0           ; show error for OpenLibrary()
                RTS
                DC.W    VERSION
                DC.W    REVISION
_PEDData:
                DC.L    printerName
                DC.L    _Init
                DC.L    _Expunge
                DC.L    _Open
                DC.L    _Close
                DC.B    PPC_COLORGFX    ;PrinterClass
                DC.B    PCC_YMCB        ; ColorClass
                DC.B    136             ; MaxColumns
                DC.B    10              ; NumCharSets
                DC.W    8               ; NumRows
                DC.L    1632            ; MaxXDots
                DC.L    0               ; MaxYDots
                DC.W    120             ; XDotsInch
                DC.W    72              ; YDotsInch
                DC.L    _CommandTable   ; Commands
                DC.L    _DoSpecial
                DC.L    _Render
                DC.L    30              ; Timeout
                DC.L    _ExtendedCharTable      ; 8BitChars
                DS.L    1               ; PrintMode (reserve space)
                DC.L    0               ; ptr to char conversion function

printerName:
                dc.b    'EpsonX',0

                END
```

Epsonx: epsonx_rev.i

```
VERSION         EQU     35
REVISION        EQU     1
```

Epsonx: init.asm

```
**********************************************************************
*
*       printer device functions
*
**********************************************************************

        SECTION         printer

*------ Included Files -----------------------------------------------

        INCLUDE         "exec/types.i"
        INCLUDE         "exec/nodes.i"
        INCLUDE         "exec/lists.i"
        INCLUDE         "exec/memory.i"
        INCLUDE         "exec/ports.i"
        INCLUDE         "exec/libraries.i"

        INCLUDE         "macros.i"

*------ Imported Functions -------------------------------------------

        XREF_EXE        CloseLibrary
        XREF_EXE        OpenLibrary
        XREF            _AbsExecBase

        XREF            _PEDData

*------ Exported Globals ---------------------------------------------

        XDEF            _Init
        XDEF            _Expunge
        XDEF            _Open
        XDEF            _Close
        XDEF            _PD
        XDEF            _PED
        XDEF            _SysBase
        XDEF            _DOSBase
        XDEF            _GfxBase
        XDEF            _IntuitionBase


**********************************************************************
        SECTION         printer,DATA
_PD             DC.L    0
_PED            DC.L    0
_SysBase        DC.L    0
_DOSBase        DC.L    0
_GfxBase        DC.L    0
_IntuitionBase  DC.L    0


**********************************************************************
        SECTION         printer,CODE
_Init:
                MOVE.L  4(A7),_PD
                LEA     _PEDData(PC),A0
                MOVE.L  A0,_PED
                MOVE.L  A6,-(A7)
                MOVE.L  _AbsExecBase,A6
                MOVE.L  A6,_SysBase

*           ;------ open the dos library
                LEA     DLName(PC),A1
                MOVEQ   #0,D0
                CALLEXE OpenLibrary
                MOVE.L  D0,_DOSBase
                BEQ     initDLErr
```

```
*           ;------ open the graphics library
                LEA     GLName(PC),A1
                MOVEQ   #0,D0
                CALLEXE OpenLibrary
                MOVE.L  D0,_GfxBase
                BEQ     initGLErr

*           ;------ open the intuition library
                LEA     ILName(PC),A1
                MOVEQ   #0,D0
                CALLEXE OpenLibrary
                MOVE.L  D0,_IntuitionBase
                BEQ     initILErr

                MOVEQ   #0,D0
pdiRts:
                MOVE.L  (A7)+,A6
                RTS

initPAErr:
                MOVE.L  _IntuitionBase,A1
                LINKEXE CloseLibrary

initILErr:
                MOVE.L  _GfxBase,A1
                LINKEXE CloseLibrary

initGLErr:
                MOVE.L  _DOSBase,A1
                LINKEXE CloseLibrary

initDLErr:
                MOVEQ   #-1,D0
                BRA.S   pdiRts

ILName:
                DC.B    'intuition.library'
                DC.B    0
DLName:
                DC.B    'dos.library'
                DC.B    0
GLName:
                DC.B    'graphics.library'
                DC.B    0
                DS.W    0


*-----------------------------------------------------------------------
_Expunge:
                MOVE.L  _IntuitionBase,A1
                LINKEXE CloseLibrary

                MOVE.L  _GfxBase,A1
                LINKEXE CloseLibrary

                MOVE.L  _DOSBase,A1
                LINKEXE CloseLibrary


*-----------------------------------------------------------------------
_Open:
                MOVEQ   #0,D0
                RTS


*-----------------------------------------------------------------------
_Close:
                MOVEQ   #0,D0
                RTS

                END
```

Epsonx: data.c

```
/*
        Data.c table for EpsonX driver.
*/

char *CommandTable[] ={
        "\375\033@\375",/* 00 aRIS reset                    */
        "\377",         /* 01 aRIN initialize               */
        "\012",         /* 02 aIND linefeed                 */
        "\015\012",     /* 03 aNEL CRLF                     */
        "\377",         /* 04 aRI reverse LF                */

                        /* 05 aSGR0 normal char set         */
        "\0335\033-\376\033F",
        "\0334",        /* 06 aSGR3 italics on              */
        "\0335",        /* 07 aSGR23 italics off            */
        "\033-\001",    /* 08 aSGR4 underline on            */
        "\033-\376",    /* 09 aSGR24 underline off          */
        "\033E",        /* 10 aSGR1 boldface on             */
        "\033F",        /* 11 aSGR22 boldface off           */
        "\377",         /* 12 aSFC set foreground color     */
        "\377",         /* 13 aSBC set background color     */

                        /* 14 aSHORP0 normal pitch          */
        "\033P\022\033W\376",
                        /* 15 aSHORP2 elite on              */
        "\033M\022\033W\376",
        "\033P",        /* 16 aSHORP1 elite off             */
                        /* 17 aSHORP4 condensed fine on     */
        "\017\033P\033W\376",
        "\022",         /* 18 aSHORP3 condensed fine off    */
        "\033W\001",    /* 19 aSHORP6 enlarge on            */
        "\033W\376",    /* 20 aSHORP5 enlarge off           */

        "\377",         /* 21 aDEN6 shadow print on         */
        "\377",         /* 22 aDEN5 shadow print off        */
        "\033G",        /* 23 aDEN4 double strike on        */
        "\033H",        /* 24 aDEN3 double strike off       */
        "\033x\001",    /* 25 aDEN2 NLQ on                  */
        "\033x\376",    /* 26 aDEN1 NLQ off                 */

        "\033S\376",    /* 27 aSUS2 superscript on          */
        "\033T",        /* 28 aSUS1 superscript off         */
        "\033S\001",    /* 29 aSUS4 subscript on            */
        "\033T",        /* 30 aSUS3 subscript off           */
        "\033T",        /* 31 aSUS0 normalize the line      */
        "\377",         /* 32 aPLU partial line up          */
        "\377",         /* 33 aPLD partial line down        */

        "\033R\376",    /* 34 aFNT0 Typeface 0              */
        "\033R\001",    /* 35 aFNT1 Typeface 1              */
        "\033R\002",    /* 36 aFNT2 Typeface 2              */
        "\033R\003",    /* 37 aFNT3 Typeface 3              */
        "\033R\004",    /* 38 aFNT4 Typeface 4              */
        "\033R\005",    /* 39 aFNT5 Typeface 5              */
        "\033R\006",    /* 40 aFNT6 Typeface 6              */
        "\033R\007",    /* 41 aFNT7 Typeface 7              */
        "\033R\010",    /* 42 aFNT8 Typeface 8              */
        "\033R\011",    /* 43 aFNT9 Typeface 9              */
        "\033R\012",    /* 44 aFNT10 Typeface 10            */

        "\033p1",       /* 45 aPROP2 proportional on        */
        "\033p0",       /* 46 aPROP1 proportional off       */
        "\377",         /* 47 aPROP0 proportional clear     */
        "\377",         /* 48 aTSS set proportional offset  */
        "\377",         /* 49 aJFY5 auto left justify       */
        "\377",         /* 50 aJFY7 auto right justify      */
        "\377",         /* 51 aJFY6 auto full jusitfy       */
        "\377",         /* 52 aJFY0 auto jusity off         */
        "\377",         /* 53 aJFY3 letter space            */
        "\377",         /* 54 aJFY1 word fill               */

        "\0330",        /* 55 aVERP0 1/8" line spacing      */
        "\0332",        /* 56 aVERP1 1/6" line spacing      */
        "\033C",        /* 57 aSLPP set form length         */
        "\033N",        /* 58 aPERF perf skip n (n > 0)     */
        "\033O",        /* 59 aPERF0 perf skip off           */

        "\377",         /* 60 aLMS set left margin           */
        "\377",         /* 61 aRMS set right margin          */
        "\377",         /* 62 aTMS set top margin            */
        "\377",         /* 63 aBMS set bottom margin         */
        "\377",         /* 64 aSTBM set T&B margins          */
        "\377",         /* 65 aSLRM set L&R margins          */
        "\377",         /* 66 aCAM clear margins             */

        "\377",         /* 67 aHTS set horiz tab             */
        "\377",         /* 68 aVTS set vert tab              */
        "\377",         /* 69 aTBC0 clear horiz tab          */
        "\033D\376",    /* 70 aTBC3 clear all horiz tabs     */
        "\377",         /* 71 aTBC1 clear vert tab           */
        "\033B\376",    /* 72 aTBC4 clear all vert tabs      */
                        /* 73 aTBCALL clear all h & v tabs   */
        "\033D\376\033B\376",
                        /* 74 aTBSALL set default tabs       */
        "\033D\010\020\030\040\050\060\070\100\110\120\130\376",

        "\377",         /* 75 aEXTEND extended commands      */
        "\377"          /* 76 aRAW next 'n' chars are raw    */
};

/*
   For each character from character 160 to character 255, there is
   an entry in this table, which is used to print (or simulate printing of)
   the full Amiga character set. (see AmigaDos Developer's Manual, pp A-3)
   This table is used only if there is a valid pointer to this table
   in the PEDData table in the printertag.asm file, and the VERSION is
   33 or greater.  Otherwise, a default table is used instead.
   To place non-printable characters in this table, you can either enter
   them as in C strings (ie \011, where 011 is an octal number, or as
   \\000 where 000 is any decimal number, from 1 to 3 digits.  This is
   usually used  to enter a NUL into the array (C has problems with it
   otherwise.), or if you forgot your octal calculator.  On retrospect,
   was a poor choice for this function, as you must say \\\\ to enter a
   backslash as a backslash.  Live and learn...
*/
char *ExtendedCharTable[] = {
        " ",                                    /* NBSP*/
        "\033R\007[\033R\\0",                   /* i */
        "c\010|",                               /* c| */
        "\033R\003#\033R\\0",                   /* L- */
        "\033R\005$\033R\\0",                   /* o */
        "\033R\010\\\\\033R\\0",                /* Y- */
        "|",                                    /* | */
        "\033R\002@\033R\\0",                   /* SS */

        "\033R\001~\033R\\0",                   /* " */
        "c",                                    /* copyright */
        "\033S\\0a\010_\033T",                  /* a_ */
        "<",                                    /* << */
        "~",                                    /* - */
        "-",                                    /* SHY */
        "r",                                    /* registered trademark */
        "-",                                    /* - */

        "\033R\001[\033R\\0",                   /* degrees */
        "+\010_",                               /* +_ */
        "\033S\\0002\033T",                     /* 2 */
        "\033S\\0003\033T",                     /* 3 */
        "'",                                    /* ' */
        "u",                                    /* u */
        "P",                                    /* reverse P */
        "\033S\\000.\033T",                     /* . */

        ",",                                    /* , */
        "\033S\\0001\033T",                     /* 1 */
        "\033R\001[\033R\\0\010-",              /* o_ */
        ">",                                    /* >> */
        "\033S\\0001\033T\010-\010\033S\0014\033T",     /* 1/4 */
        "\033S\\0001\033T\010-\010\033S\0012\033T",     /* 1/2 */
        "\033S\\0003\033T\010-\010\033S\0014\033T",     /* 3/4 */
        "\033R\007]\033R\\0",                   /* upside down ? */
        "A\010;                                 /* À */
        "A\010'",                                 /* 'A */
        "A\010^",                                 /* ^A */
        "A\010~",                                 /* ~A */
        "\033R\002[\033R\\0",                    /* "A */
        "\033R\004]\033R\\0",                    /* oA */
        "\033R\004[\033R\\0",                    /* AE */
        "C\010,",                                 /* C, */

        "E\010;                                 /* È */
        "\033R\011@\033R\\0",                    /* 'E */
        "E\010^",                                 /* ^E */
        "E\010\033R\001~\033R\\0",               /* "E */
        "I\010;                                 /* Ì */
        "I\010;                                 /* 'I */
        "I\010^",                                 /* ^I */
        "I\010\033R\001~\033R\\0",               /* "I */

        "D\010-",                                 /* -D */
        "\033R\007\\\\\033R\\0",                 /* ~N */
        "O\010;                                 /* Ò */
        "O\010'",                                 /* 'O */
        "O\010^",                                 /* ^O */
        "O\010~",                                 /* ~O */
        "\033R\002\\\\\033R\\0",                 /* "O */
        "x",                                      /* x */

        "\033R\004\\\\\033R\\0",                 /* 0 */
        "U\010;                                 /* Ù */
        "U\010'",                                 /* 'U */
        "U\010^",                                 /* ^U */
        "\033R\002]\033R\\0",                    /* "U */
        "Y\010'",                                 /* 'Y */
        "T",                                      /* Thorn */
        "\033R\002~\033R\\0",                    /* B */

        "\033R\001@\033R\\0",                    /* à */
        "a\010'",                                 /* 'a */
        "a\010^",                                 /* ^a */
        "a\010~",                                 /* ~a */
        "\033R\002{\033R\\0",                    /* "a */
        "\033R\004}\033R\\0",                    /* oa */
        "\033R\004{\033R\\0",                    /* ae */
        "\033R\001\\\\\033R\\0",                 /* c, */

        "\033R\001}\033R\\0",                    /* è */
        "\033R\001{\033R\\0",                    /* 'e */
        "e\010^",                                 /* ^e */
        "e\010\033R\001~\033R\\0",               /* "e */
        "\033R\006~\033R\\0",                    /* ì */
        "i\010'",                                 /* 'i */
        "i\010^",                                 /* ^i */
        "i\010\033R\001~\033R\\0",               /* "i */

        "d",                                      /* d */
        "\033R\007|\033R\\0",                    /* ~n */
        "\033R\006|\033R\\0",                    /* ò */
        "o\010'",                                 /* 'o */
        "o\010^",                                 /* ^o */
        "o\010~",                                 /* ~o */
        "\033R\002|\033R\\0",                    /* "o */
        ":\010-"                                  /* :- */

        "\033R\004|\033R\\0",                    /* o/ */
        "\033R\001|\033R\\0",                    /* ù */
        "u\010'",                                 /* 'u */
        "u\010^",                                 /* ^u */
        "\033R\002}\033R\\0",                    /* "u */
        "y\010'",                                 /* 'y */
        "t",                                      /* thorn */
        "y\010\033R\001~\033R\\0"                /* "y */
};
```

Epsonx: dospecial.c

```
/*
        DoSpecial for EpsonX driver.
*/

#include "exec/types.h"
#include "devices/printer.h"
#include "devices/prtbase.h"

#define LMARG   3
#define RMARG   6
#define MARGLEN 8

#define CONDENSED       7
#define PITCH           9
#define QUALITY         17
#define LPI             24
#define INITLEN         26

DoSpecial(command, outputBuffer, vline, currentVMI, crlfFlag, Parms)
char outputBuffer[];
UWORD *command;
BYTE *vline;
BYTE *currentVMI;
BYTE *crlfFlag;
UBYTE Parms[];
{
        extern struct PrinterData *PD;

        int x = 0, y = 0;
        /*
                00-00   \375    wait
                01-03   \0331L  set left margin
                04-06   \033Qq  set right margin
                07-07   \375    wait
        */

        static char initMarg[MARGLEN+1] = "\375\0331L\033Qq\375";
        /*
                00-01   \0335           italics off
                02-04   \033-\000       underline off
                05-06   \033F           boldface off
                07-07   \022            cancel condensed mode
                08-09   \033P           select pica (10 cpi)
                10-12   \033W\000       enlarge off
                13-14   \033H           doublestrike off
                15-17   \033x\000       draft
                18-19   \033T           super/sub script off
                20-22   \033p0          proportional off
                23-24   \0332           6 lpi
                25-25   \015            carriage return
        */
        static char initThisPrinter[INITLEN+1] =
        "\0335\033-\000\033F\022\033P\033W\000\033H\033x\000\033T\033p0\0332\015";

        static BYTE ISOcolorTable[10] = {0, 5, 6, 4, 3, 1, 2, 0};

        if (*command == aRIN) {
                while (x < INITLEN) {
                        outputBuffer[x] = initThisPrinter[x];
                        x++;
                }

                if (PD->pd_Preferences.PrintQuality == LETTER) {
                        outputBuffer[QUALITY] = 1;
                }

                *currentVMI = 36; /* assume 1/6 line spacing (36/216 => 1/6) */
                if (PD->pd_Preferences.PrintSpacing == EIGHT_LPI) {
                        outputBuffer[LPI] = '0';
                        *currentVMI = 27; /* 27/216 => 1/8 */
                }

                if (PD->pd_Preferences.PrintPitch == ELITE) {
                        outputBuffer[PITCH] = 'M';
                }
        else if (PD->pd_Preferences.PrintPitch == FINE) {
                outputBuffer[CONDENSED] = '\017'; /* condensed */
                outputBuffer[PITCH] = 'P'; /* pica condensed */
        }

        Parms[0] = PD->pd_Preferences.PrintLeftMargin;
        Parms[1] = PD->pd_Preferences.PrintRightMargin;
        *command = aSLRM;
}

if (*command == aCAM) { /* cancel margins */
        y = PD->pd_Preferences.PaperSize == W_TRACTOR ? 136 : 80;
        if (PD->pd_Preferences.PrintPitch == PICA) {
                Parms[1] = (10 * y) / 10;
        }
        else if (PD->pd_Preferences.PrintPitch == ELITE) {
                Parms[1] = (12 * y) / 10;
        }
        else { /* fine */
                Parms[1] = (17 * y) / 10;
        }
        Parms[0] = 1;
        y = 0;
        *command = aSLRM;
}

if (*command == aSLRM) { /* set left and right margins */
        PD->pd_PWaitEnabled = 253;
        if (Parms[0] == 0) {
                initMarg[LMARG] = 0;
        }
        else {
                initMarg[LMARG] = Parms[0] - 1;
        }
        initMarg[RMARG] = Parms[1];
        while (y < MARGLEN) {
                outputBuffer[x++] = initMarg[y++];
        }
        return(x);
}

if (*command == aPLU) {
        if (*vline == 0) {
                *vline = 1;
                *command = aSUS2;
                return(0);
        }
        if (*vline < 0) {
                *vline = 0;
                *command = aSUS3;
                return(0);
        }
        return(-1);
}

if (*command == aPLD) {
        if (*vline == 0) {
                *vline = -1;
                *command = aSUS4;
                return(0);
        }
        if (*vline > 0) {
                *vline = 0;
                *command = aSUS1;
                return(0);
        }
        return(-1);
}

if (*command == aSUS0) {
        *vline = 0;
}

if (*command == aSUS1) {
        *vline = 0;
}
```

```
        if (*command == aSUS2) {
                *vline = 1;
        }

        if (*command == aSUS3) {
                *vline = 0;
        }

        if (*command == aSUS4) {
                *vline = -1;
        }

        if (*command == aVERP0) {
                *currentVMI = 27;
        }

        if (*command == aVERP1) {
                *currentVMI = 36;
        }

        if (*command == aIND) { /* lf */
                outputBuffer[x++] = '\033';
                outputBuffer[x++] = 'J';
                outputBuffer[x++] = *currentVMI;
                return(x);
        }

        if (*command == aRI) { /* reverse lf */
                outputBuffer[x++] = '\033';
                outputBuffer[x++] = 'j';
                outputBuffer[x++] = *currentVMI;
                return(x);
        }

        if (*command == aSFC) {
                if (Parms[0] == 39) {
                        Parms[0] = 30; /* set defaults */
                }
                if (Parms[0] > 37) {
                        return(0); /* ni or background color change */
                }
                outputBuffer[x++] = '\033';
                outputBuffer[x++] = 'r';
                outputBuffer[x++] = ISOcolorTable[Parms[0] - 30];
                /*
                Kludge to get this to work on a CBM_MPS-1250  which interprets
                'ESCr' as go into reverse print mode.  The 'ESCt' tells it to
                get out of reverse print mode.  The 'NULL' is ignored by the
                CBM_MPS-1250 and required by all Epson printers as the
                terminator for the 'ESCtNULL' command which means select
                normal char set (which has no effect).
                */
                outputBuffer[x++] = '\033';
                outputBuffer[x++] = 't';
                outputBuffer[x++] = 0;
                return(x);
        }

        if (*command == aRIS) {
                PD->pd_PWaitEnabled = 253;
        }

        return(0);
}
```

Epsonx: render.c

```
/*
        EpsonX (EX/FX/JX/LX/MX/RX) driver.
*/

#include <exec/types.h>
#include <exec/nodes.h>
#include <exec/lists.h>
#include <exec/memory.h>
#include "devices/printer.h"
#include "devices/prtbase.h"

#define NUMSTARTCMD     7       /* # of cmd bytes before binary data */
#define NUMENDCMD       1       /* # of cmd bytes after binary data */
#define NUMTOTALCMD     (NUMSTARTCMD + NUMENDCMD)       /* total of above */
#define NUMLFCMD        4       /* # of cmd bytes for linefeed */
#define MAXCOLORBUFS    4       /* max # of color buffers */

#define STARTLEN        19
#define PITCH           1
#define CONDENSED       2
#define LMARG           8
#define RMARG           11
#define DIREC           15

static ULONG TwoBufSize;
static UWORD RowSize, ColorSize, NumColorBufs, dpi_code, spacing;
static UWORD colorcodes[MAXCOLORBUFS];

Render(ct, x, y, status)
long ct, x, y, status;
{
        extern void *AllocMem(), FreeMem();

        extern struct PrinterData *PD;
        extern struct PrinterExtendedData *PED;

        UBYTE *CompactBuf();
        static ULONG BufSize, TotalBufSize, dataoffset;
        static UWORD spacing, colors[MAXCOLORBUFS];
        /*
                00-01   \003P           set pitch (10 or 12 cpi)
                02-02   \022            set condensed fine (on or off)
                03-05   \033W\000       enlarge off
                06-08   \033ln          set left margin to n
                09-11   \033Qn          set right margin to n
                12-12   \015            carriage return
                13-15   \033U1          set uni-directional mode
                16-18   \033t\000       see kludge note below
                Kludge to get this to work on a CBM_MPS-1250  which interprets
                'ESCr' as go into reverse print mode.  The 'ESCt' tells it to
                get out of reverse print mode.  The 'NULL' is ignored by the
                CBM_MPS-1250 and required by all Epson printers as the
                terminator for the 'ESCtNULL' command which means select
                normal char set (which has no effect).
        */

        static UBYTE StartBuf[STARTLEN+1] =
                "\033P\022\033W\000\033ln\033Qn\015\033U1\033t\000";

        UBYTE *ptr, *ptrstart;
        int err;

        switch(status) {
                case 0 : /* Master Initialization */
                        /*
                                ct      - pointer to IODRPReq structure.
                                x       - width of printed picture in pixels.
                                y       - height of printed picture in pixels.
                        */
                        RowSize = x;
                        ColorSize = RowSize + NUMTOTALCMD;
                        if (PD->pd_Preferences.PrintShade == SHADE_COLOR) {
                                NumColorBufs = MAXCOLORBUFS;
                                colors[0] = ColorSize * 3; /* Black */
                                colors[1] = ColorSize * 0; /* Yellow */
                        colors[2] = ColorSize * 1; /* Magenta */
                        colors[3] = ColorSize * 2; /* Cyan */
                        colorcodes[0] = 4; /* Yellow */
                        colorcodes[1] = 1; /* Magenta */
                        colorcodes[2] = 2; /* Cyan */
                        colorcodes[3] = 0; /* Black */
                }
                else { /* grey-scale or black&white */
                        NumColorBufs = 1;
                        colors[0] = ColorSize * 0; /* Black */
                        colorcodes[0] = 0; /* Black */
                }
                BufSize = ColorSize * NumColorBufs + NUMLFCMD;
                if (PED->ped_YDotsInch == 216) {
                        TwoBufSize = BufSize * 3;
                        TotalBufSize = BufSize * 6;
                }
                else if (PED->ped_YDotsInch == 144) {
                        TwoBufSize = BufSize * 2;
                        TotalBufSize = BufSize * 4;
                }
                else {
                        TwoBufSize = BufSize * 1;
                        TotalBufSize = BufSize * 2;
                }
                PD->pd_PrintBuf = AllocMem(TotalBufSize, MEMF_PUBLIC);
                if (PD->pd_PrintBuf == NULL) {
                        err = PDERR_BUFFERMEMORY;
                }
                else {
                        dataoffset = NUMSTARTCMD;
                        /*
                                This printer prints graphics within its
                                text margins.  This code makes sure the
                                printer is in 10 cpi and then sets the
                                left and right margins to their minimum
                                and maximum values (respectively).  A
                                carriage return is sent so that the
                                print head is at the leftmost position
                                as this printer starts printing from
                                the print head's position.  The printer
                                is put into unidirectional mode to
                                reduce wavy vertical lines.
                        */
                        StartBuf[PITCH] = 'P'; /* 10 cpi */
                        StartBuf[CONDENSED] = '\022'; /* off */
                        /* left margin of 1 */
                        StartBuf[LMARG] = 0;
                        /* right margin of 80 or 136 */
                        StartBuf[RMARG] = PD->pd_Preferences.
                                PaperSize == W_TRACTOR ? 136 : 80;
                        /* uni-directional mode */
                        StartBuf[DIREC] = '1';
                        err = (*(PD->pd_PWrite))(StartBuf, STARTLEN);
                }
                break;

        case 1 : /* Scale, Dither and Render */
                /*
                        ct      - pointer to PrtInfo structure.
                        x       - 0.
                        y       - row # (0 to Height - 1).
                */
                Transfer(ct, y, &PD->pd_PrintBuf[dataoffset], colors,
                        BufSize);
                err = PDERR_NOERR; /* all ok */
                break;

        case 2 : /* Dump Buffer to Printer */
                /*
                        ct      - 0.
                        x       - 0.
                        y       - # of rows sent (1 to NumRows).

                */
                /* white-space strip */
                ptrstart = &PD->pd_PrintBuf[dataoffset - NUMSTARTCMD];
                if (PED->ped_YDotsInch == 72) {
                        /* y range : 1 to 8 */
                        y = y * 3 - spacing;
                        ptr = CompactBuf(ptrstart + NUMSTARTCMD,
                                ptrstart, y, 1);
                }
                else if (PED->ped_YDotsInch == 144) {
                        /* y range : 1 to 16 */
                        ptr = CompactBuf(ptrstart + NUMSTARTCMD,
                                ptrstart, 2, 1);
                        if (y > 1) {
                                ptr = CompactBuf(&PD->pd_PrintBuf[
                                        dataoffset + BufSize],
                                        ptr, y * 3 / 2 - 2, 0);
                        }
                }
                else if (PED->ped_YDotsInch == 216) {
                        /* y range : 1 to 24 */
                        ptr = CompactBuf(ptrstart + NUMSTARTCMD,
                                ptrstart, 1, 1);
                        if (y > 1) {
                                ptr = CompactBuf(&PD->pd_PrintBuf[
                                        dataoffset + BufSize],
                                        ptr, 1, 0);
                        }
                        if (y > 2) {
                                ptr = CompactBuf(&PD->pd_PrintBuf[
                                        dataoffset + BufSize * 2],
                                        ptr, y - 2, 0);
                        }
                }
                err = (*(PD->pd_PWrite))(ptrstart, ptr - ptrstart);
                if (err == PDERR_NOERR) {
                        dataoffset = (dataoffset == NUMSTARTCMD ?
                                TwoBufSize : 0) + NUMSTARTCMD;
                }
                break;

        case 3 : /* Clear and Init Buffer */
                /*
                        ct      - 0.
                        x       - 0.
                        y       - 0.
                */
                ClearAndInit(&PD->pd_PrintBuf[dataoffset]);
                err = PDERR_NOERR;
                break;

        case 4 : /* Close Down */
                /*
                        ct      - error code.
                        x       - io_Special flag from IODRPReq.
                        y       - 0.
                */
                err = PDERR_NOERR; /* assume all ok */
                /* if user did not cancel print */
                if (ct != PDERR_CANCEL) {
                        /* restore preferences pitch and margins */
                        if (PD->pd_Preferences.PrintPitch == ELITE) {
                                StartBuf[PITCH] = 'M'; /* 12 cpi */
                        }
                        else if (PD->pd_Preferences.PrintPitch == FINE) {
                                StartBuf[CONDENSED] = '\017'; /* on */
                        }
                        StartBuf[LMARG] =
                                PD->pd_Preferences.PrintLeftMargin - 1;
                        StartBuf[RMARG] =
                                PD->pd_Preferences.PrintRightMargin;
                        StartBuf[DIREC] = '0'; /* bi-directional */
                        err = (*(PD->pd_PWrite))(StartBuf, STARTLEN);
                }
                (*(PD->pd_PBothReady))();
                if (PD->pd_PrintBuf != NULL) {
                        FreeMem(PD->pd_PrintBuf, TotalBufSize);
                }
                break;
                case 5 :  /* Pre-Master Initialization */
                        /*
                                ct      - 0 or pointer to IODRPReq structure.
                                x       - io_Special flag from IODRPReq.
                                y       - 0.
                        */
                        /* kludge for sloppy tractor mechanism */
                        spacing = PD->pd_Preferences.PaperType == SINGLE ?
                                1 : 0;
                        dpi_code = SetDensity(x & SPECIAL_DENSITYMASK);
                        err = PDERR_NOERR;
                        break;
        }
        return(err);
}


UBYTE *CompactBuf(ptrstart, ptr2start, y, flag)
UBYTE *ptrstart, *ptr2start;
long y;
int flag; /* 0 - not first pass, !0 - first pass */
{
        static int x;
        UBYTE *ptr, *ptr2;
        long ct;
        int i;

        ptr2 = ptr2start; /* where to put the compacted data */
        if (flag) {
                x = 0; /* flag no transfer required yet */
        }

        for (ct=0; ct<NumColorBufs; ct++, ptrstart += ColorSize) {
                i = RowSize;
                ptr = ptrstart + i - 1;
                while (i > 0 && *ptr == 0) {
                        i--;
                        ptr--;
                }

                if (i != 0) { /* if data */
                        *(++ptr) = 13;                  /* <cr> */
                        ptr = ptrstart - NUMSTARTCMD;
                        *ptr++ = 27;
                        *ptr++ = 'r';
                        *ptr++ = colorcodes[ct];        /* color */
                        *ptr++ = 27;
                        *ptr++ = dpi_code;              /* density */
                        *ptr++ = i & 0xff;
                        *ptr++ = i >> 8;                 /* size */
                        i += NUMTOTALCMD;
                        if (x != 0) { /* if must transfer data */
                                /* get src start */
                                ptr = ptrstart - NUMSTARTCMD;
                                do { /* transfer and update dest ptr */
                                        *ptr2++ = *ptr++;
                                } while (--i);
                        }

                        else { /* no transfer required */
                                ptr2 += i; /* update dest ptr */
                        }
                }

                if (i != RowSize + NUMTOTALCMD) { /* if compacted or 0 */
                        x = 1; /* flag that we need to transfer next time */
                }
        }

        *ptr2++ = 13; /* cr */
        *ptr2++ = 27;
        *ptr2++ = 'J';
        *ptr2++ = y; /* y/216 lf */
        return(ptr2);
}
```

```
ClearAndInit(ptr)
UBYTE *ptr;
{
        ULONG *lptr, i, j;

        /*
                Note : Since 'NUMTOTALCMD + NUMLFCMD' is > 3 bytes if is safe
                to do the following to speed things up.
        */
        i = TwoBufSize - NUMTOTALCMD - NUMLFCMD;
        j = (ULONG)ptr;
        if (!(j & 1)) { /* if on a word boundary, clear by longs */
                i = (i + 3) / 4;
                lptr = (ULONG *)ptr;
                do {
                        *lptr++ = 0;
                } while (--i);
        }
        else { /* clear by bytes */
                do {
                        *ptr++ = 0;
                } while (--i);
        }
        return(0);
}
```

Epsonx: transfer.asm

```
**********************************************************************
*
* Transfer routine for EpsonX
*
**********************************************************************

        INCLUDE "exec/types.i"

        INCLUDE "intuition/intuition.i"
        INCLUDE "devices/printer.i"
        INCLUDE "devices/prtbase.i"
        INCLUDE "devices/prtgfx.i"

        XREF    _PD
        XREF    _PED
        XREF    _LVODebug
        XREF    _AbsExecBase

        XDEF    _Transfer

        SECTION         printer,CODE
_Transfer:
; Transfer(PInfo, y, ptr, colors, BufOffset)
; struct PrtInfo *PInfo         4-7
; UWORD y;                      8-11
; UBYTE *ptr;                   12-15
; UWORD *colors;                16-19
; ULONG BufOffset               20-23

        movem.l d2-d6/a2-a4,-(sp)       ;save regs used

        movea.l 36(sp),a0               ;a0 = PInfo
        move.l  40(sp),d0               ;d0 = y
        movea.l 44(sp),a1               ;a1 = ptr
        movea.l 48(sp),a2               ;a2 = colors
        move.l  52(sp),d1               ;d1 = BufOffset

        move.l  d0,d3                   ;save y
        moveq.l #3,d2
        and.w   d0,d2                   ;d2 = y & 3
        lsl.w   #2,d2                   ;d2 = (y & 3) << 2
        movea.l pi_dmatrix(a0),a3       ;a3 = dmatrix
        adda.l  d2,a3                   ;a3 = dmatrix + ((y & 3) << 2)

        movea.l _PED,a4                 ;a4 = ptr to PED
        cmpi.w  #216,ped_YDotsInch(a4)  ;triple interleaving?
        bne.s   10$                     ;no
        divu.w  #3,d0                   ;y /= 3
        swap.w  d0                      ;d0 = y % 3
        mulu.w  d0,d1                   ;BufOffset *= y % 3
        swap.w  d0                      ;d0 = y / 3
        bra.s   30$

10$:    cmpi.w  #144,ped_YDotsInch(a4)  ;double interleaving?
        bne.s   20$                     ;no, clear BufOffset
        asr.w   #1,d0                   ;y /= 2
        btst    #0,d3                   ;odd pass?
        bne.s   30$                     ;no, dont clear BufOffset

20$:    moveq.l #0,d1                   ;BufOffset = 0

30$:    move.w  d0,d6
        not.b   d6                      ;d6 = bit to set
        adda.l  d1,a1                   ;ptr += BufOffset

        movea.l _PD,a4                  ;a4 = ptr to PD
        cmpi.w  #SHADE_COLOR,pd_Preferences+pf_PrintShade(a4)   ;color dump?
        bne     not_color               ;no

color:

; a0 - PInfo
; a1 - ptr (ptr + BufOffset)
; a2 - colors
; a3 - dmatrix ptr
; d0 - y
; d1 - BufOffset
; d6 - bit to set

        movem.l d7/a5-a6,-(sp)          ;save regs used

        movea.l a1,a4
        movea.l a1,a5
        movea.l a1,a6
        adda.w  (a2)+,a1                ;a1 = ptr + colors[0] (bptr)
        adda.w  (a2)+,a4                ;a4 = ptr + colors[1] (yptr)
        adda.w  (a2)+,a5                ;a5 = ptr + colors[2] (mptr)
        adda.w  (a2)+,a6                ;a6 = ptr + colors[3] (cptr)

;       move.l  a6,-(sp)
;       move.l  _AbsExecBase,a6
;       jsr     _LVODebug(a6)
;       move.l  (sp)+,a6

        movea.l pi_ColorInt(a0),a2      ;a2 = ColorInt ptr
        move.w  pi_width(a0),width      ;# of pixels to do
        move.w  pi_xpos(a0),d2          ;d2 = x
        movea.l pi_ScaleX(a0),a0        ;a0 = ScaleX (sxptr)
        move.b  d6,d7                   ;d7 = bit to set

; a0 - sxptr
; a1 - bptr
; a2 - ColorInt ptr
; a3 - dmatrix ptr
; a4 - yptr
; a5 - mptr
; a6 - cptr
; d1 - Black
; d2 - x
; d3 - dvalue (dmatrix[x & 3])
; d4 - Yellow
; d5 - Magenta
; d6 - Cyan
; d7 - bit to set

cwidth_loop:
        move.b  PCMBLACK(a2),d1         ;d1 = Black
        move.b  PCMYELLOW(a2),d4        ;d4 = Yellow
        move.b  PCMMAGENTA(a2),d5       ;d5 = Magenta
        move.b  PCMCYAN(a2),d6          ;d6 = Cyan
        addq.l  #ce_SIZEOF,a2           ;advance to next entry

        move.w  (a0)+,sx                ;# of times to use this pixel
```

```
csx_loop:
        moveq.l #3,d3
        and.w   d2,d3                       ;d3 = x & 3
        move.b  0(a3,d3.w),d3               ;d3 = dmatrix[x & 3]

black:
        cmp.b   d3,d1                       ;render black?
        ble.s   yellow                      ;no, try ymc
        bset.b  d7,0(a1,d2.w)               ;set black pixel
        bra.s   csx_end

yellow:
        cmp.b   d3,d4                       ;render yellow pixel?
        ble.s   magenta                     ;no.
        bset.b  d7,0(a4,d2.w)               ;set yellow pixel

magenta:
        cmp.b   d3,d5                       ;render magenta pixel?
        ble.s   cyan                        ;no.
        bset.b  d7,0(a5,d2.w)               ;set magenta pixel

cyan:
        cmp.b   d3,d6                       ;render cyan pixel?
        ble.s   csx_end                     ;no, skip to next pixel.
        bset.b  d7,0(a6,d2.w)               ;clear cyan pixel

csx_end:
        addq.w  #1,d2                       ;x++
        subq.w  #1,sx                       ;sx--
        bne.s   csx_loop
        subq.w  #1,width                    ;width--
        bne.s   cwidth_loop

        movem.l (sp)+,d7/a5-a6              ;restore regs used
        bra     exit

not_color:
; a0 - PInfo
; a1 - ptr
; a2 - colors
; a3 - dmatrix ptr
; d0 - y
; d6 - bit to set

        adda.w  (a2),a1                     ;a1 = ptr + colors[0]
        move.w  pi_width(a0),d1             ;d1 = width
        subq.w  #1,d1                       ;adjust for dbra

        move.w  pi_threshold(a0),d3         ;d3 = threshold, thresholding?
        beq.s   grey_scale                  ;no, grey-scaling

threshold:
; a0 - PInfo
; a1 - ptr
; a3 - dmatrix ptr
; d1 - width-1
; d3 - threshold
; d6 - bit to set

        eori.b  #15,d3                      ;d3 = dvalue
        movea.l pi_ColorInt(a0),a2          ;a2 = ColorInt ptr
        move.w  pi_xpos(a0),d2              ;d2 = x
        movea.l pi_ScaleX(a0),a0            ;a0 = ScaleX (sxptr)
        adda.w  d2,a1                       ;ptr += x

; a0 - sxptr
; a1 - ptr
; a2 - ColorInt ptr
; a3 - dmatrix ptr (NOT USED)
; d1 - width
; d3 - dvalue
; d4 - Black
; d5 - sx
; d6 - bit to set

twidth_loop:
        move.b  PCMBLACK(a2),d4             ;d4 = Black
        addq.l  #ce_SIZEOF,a2           ;advance to next entry

        move.w  (a0)+,d5                ;d5 = # of times to use this pixel

        cmp.b   d3,d4                   ;render this pixel?
        ble.s   tsx_end                 ;no, skip to next pixel.
        subq.w  #1,d5                   ;adjust for dbra

tsx_render:                             ;yes, render this pixel sx times
        bset.b  d6,(a1)                 ;*(ptr) |= bit;

        adda.w  #1,a1                   ;ptr++
        dbra    d5,tsx_render           ;sx--
        dbra    d1,twidth_loop          ;width--
        bra.s   exit                    ;all done

tsx_end:
        adda.w  d5,a1                   ;ptr += sx
        dbra    d1,twidth_loop          ;width--
        bra.s   exit

grey_scale:
; a0 - PInfo
; a1 - ptr
; a3 - dmatrix ptr
; d0 - y
; d1 - width-1
; d6 - bit to set

        movea.l pi_ColorInt(a0),a2      ;a2 = ColorInt ptr
        move.w  pi_xpos(a0),d2          ;d2 = x
        movea.l pi_ScaleX(a0),a0        ;a0 = ScaleX (sxptr)

; a0 - sxptr
; a1 - ptr
; a2 - ColorInt ptr
; a3 - dmatrix ptr
; d1 - width
; d2 - x
; d3 - dvalue (dmatrix[x & 3])
; d4 - Black
; d5 - sx
; d6 - bit to set

gwidth_loop:
        move.b  PCMBLACK(a2),d4         ;d4 = Black
        addq.l  #ce_SIZEOF,a2           ;advance to next entry

        move.w  (a0)+,d5                ;d5 = # of times to use this pixel
        subq.w  #1,d5                   ;adjust for dbra

gsx_loop:
        moveq.l #3,d3
        and.w   d2,d3                   ;d3 = x & 3
        move.b  0(a3,d3.w),d3           ;d3 = dmatrix[x & 3]

        cmp.b   d3,d4                   ;render this pixel?
        ble.s   gsx_end                 ;no, skip to next pixel.

        bset.b  d6,0(a1,d2.w)           ;*(ptr + x) |= bit

gsx_end
        addq.w  #1,d2                   ;x++
        dbra    d5,gsx_loop             ;sx--
        dbra    d1,gwidth_loop          ;width--

exit:
        movem.l (sp)+,d2-d6/a2-a4       ;restore regs used
        moveq.l #0,d0                   ;flag all ok
        rts                             ;goodbye

sx      dc.w    0
width   dc.w    0

        END
```

Epsonx: transfer.c

```
/*
        C-language Transfer routine for EpsonX driver.
 */

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>
#include <devices/prtgfx.h>

Transfer(PInfo, y, ptr, colors, BufOffset)
struct PrtInfo *PInfo;
UWORD y;                /* row # */
UBYTE *ptr;             /* ptr to buffer */
UWORD *colors;          /* indexes to color buffers */
ULONG BufOffset;        /* used for interleaved printing */
{
        extern struct PrinterData *PD;
        extern struct PrinterExtendedData *PED;

        static UWORD bit_table[8] = {128, 64, 32, 16, 8, 4, 2, 1};
        union colorEntry *ColorInt;
        UBYTE *bptr, *yptr, *mptr, *cptr, Black, Yellow, Magenta, Cyan;
        UBYTE *dmatrix, dvalue, threshold;
        UWORD x, width, sx, *sxptr, color, bit, x3;

        /* printer non-specific, MUST DO FOR EVERY PRINTER */
        x = PInfo->pi_xpos;
        ColorInt = PInfo->pi_ColorInt;
        sxptr = PInfo->pi_ScaleX;
        width = PInfo->pi_width;

        /* printer specific */
        if (PED->ped_YDotsInch == 216)
        {
                BufOffset *= y % 3;
                y /= 3;
        }
        else if (PED->ped_YDotsInch == 144)
        {
                BufOffset *= y & 1;
                y /= 2;
        }
        else
        {
                BufOffset = 0;
        }
        bit = bit_table[y & 7];
        bptr = ptr + colors[0] + BufOffset;
        yptr = ptr + colors[1] + BufOffset;
        mptr = ptr + colors[2] + BufOffset;
        cptr = ptr + colors[3] + BufOffset;

        /* pre-compute threshold; are we thresholding? */
        if (threshold = PInfo->pi_threshold)
        { /* thresholding */
                dvalue = threshold ^ 15;
                bptr += x;
                do { /* for all source pixels */
                        /* pre-compute intensity values for Black component */
                        Black = ColorInt->colorByte[PCMBLACK];
                        ColorInt++;

                        sx = *sxptr++;

                        do { /* use this pixel 'sx' times */
                                if (Black > dvalue)
                                {
                                        *bptr |= bit;
                                }
                                bptr++; /* done 1 more printer pixel */
                        } while (--sx);
                } while (--width);
        }

        else
        { /* not thresholding, pre-compute ptr to dither matrix */
                dmatrix = PInfo->pi_dmatrix + ((y & 3) << 2);
                if (PD->pd_Preferences.PrintShade == SHADE_GREYSCALE)
                {
                        do { /* for all source pixels */
                                /* pre-compute intensity values for Black */
                                Black = ColorInt->colorByte[PCMBLACK];
                                ColorInt++;

                                sx = *sxptr++;

                                do { /* use this pixel 'sx' times */
                                        if (Black > dmatrix[x & 3])
                                        {
                                                *(bptr + x) |= bit;
                                        }
                                        x++; /* done 1 more printer pixel */
                                } while (--sx);
                        } while (--width);
                }
                else
                { /* color */
                        do { /* for all source pixels */
                                /* compute intensity values for each color */
                                Black = ColorInt->colorByte[PCMBLACK];
                                Yellow = ColorInt->colorByte[PCMYELLOW];
                                Magenta = ColorInt->colorByte[PCMMAGENTA];
                                Cyan = ColorInt->colorByte[PCMCYAN];
                                ColorInt++;

                                sx = *sxptr++;

                                do { /* use this pixel 'sx' times */
                                        x3 = x >> 3;
                                        dvalue = dmatrix[x & 3];
                                        if (Black > dvalue)
                                        {
                                                *(bptr + x) |= bit;
                                        }
                                        else
                                        { /* black not rendered */
                                                if (Yellow > dvalue)
                                                {
                                                        *(yptr + x) |= bit;
                                                }
                                                if (Magenta > dvalue)
                                                {
                                                        *(mptr + x) |= bit;
                                                }
                                                if (Cyan > dvalue)
                                                {
                                                        *(cptr + x) |= bit;
                                                }
                                        }
                                        ++x; /* done 1 more printer pixel */
                                } while (--sx);
                        } while (--width);
                }
        }
}
```

Epsonx: density.c

```
/*
        Density module for EpsonX driver.
*/

#include <exec/types.h>
#include "devices/printer.h"
#include "devices/prtbase.h"

SetDensity(density_code)
ULONG density_code;
{
        extern struct PrinterData *PD;
        extern struct PrinterExtendedData *PED;

        /* SPECIAL_DENSITY     0    1    2    3    4    5    6    7 */
        static int XDPI[8] = {120, 120, 120, 240, 120, 240, 240, 240};
        static int YDPI[8] = {72, 72, 144, 72, 216, 144, 216, 216};
        static char codes[8] = {'L', 'L', 'L', 'Z', 'L', 'Z', 'Z', 'Z'};

        PED->ped_MaxColumns =
                PD->pd_Preferences.PaperSize == W_TRACTOR ? 136 : 80;
        density_code /= SPECIAL_DENSITY1;
        /* default is 80 chars (8.0 in.), W_TRACTOR is 136 chars (13.6 in.) */
        PED->ped_MaxXDots =
                (XDPI[density_code] * PED->ped_MaxColumns) / 10;
        PED->ped_XDotsInch = XDPI[density_code];
        PED->ped_YDotsInch = YDPI[density_code];
        if ((PED->ped_YDotsInch = YDPI[density_code]) == 216) {
                PED->ped_NumRows = 24;
        }
        else if (PED->ped_YDotsInch == 144) {
                PED->ped_NumRows = 16;
        }
        else {
                PED->ped_NumRows = 8;
        }
        return((int)codes[density_code]);
}
```

HP_Laserjet

The driver for the HP_LaserJet can be generated with the following Makefile.

```
LC = lc:lc
ASM = lc:asm
CFLAGS = -iINCLUDE: -b0 -d0 -v
ASMFLAGS = -iINCLUDE:
LINK = lc:blink
LIB = lib:amiga.lib+lib:lc.lib
OBJ = printertag.o+init.o+data.o+dospecial.o+render.o+transfer.o+density.o
TARGET = hp_laserjet

        @$(LC) $(CFLAGS) $*

$(TARGET): printertag.o init.o data.o dospecial.o render.o density.o transfer.o
        @$(LINK) <WITH <
        FROM $(OBJ)
        TO $(TARGET)
        LIBRARY $(LIB)
        NODEBUG SC SD VERBOSE MAP $(TARGET).map H
        <

init.o: init.asm
        @$(ASM) $(ASMFLAGS) init.asm

printertag.o: printertag.asm hp_rev.i
        @$(ASM) $(ASMFLAGS) printertag.asm

transfer.o: transfer.asm
        @$(ASM) $(ASMFLAGS) transfer.asm

dospecial.o: dospecial.c

data.o: data.c

density.o: density.c

render.o: render.c
```

HP_Laserjet: macros.i

```
**********************************************************************
*
*       printer device macro definitions
*
**********************************************************************

*------ external definition macros ----------------------------------

XREF_EXE        MACRO
        XREF            _LVO\1
                ENDM

XREF_DOS        MACRO
        XREF            _LVO\1
                ENDM

XREF_GFX        MACRO
        XREF            _LVO\1
                ENDM

XREF_ITU        MACRO
        XREF            _LVO\1
                ENDM

*------ library dispatch macros -------------------------------------

CALLEXE         MACRO
                CALLLIB _LVO\1
                ENDM

LINKEXE         MACRO
                LINKLIB _LVO\1,_SysBase
                ENDM

LINKDOS         MACRO
                LINKLIB _LVO\1,_DOSBase
                ENDM

LINKGFX         MACRO
                LINKLIB _LVO\1,_GfxBase
                ENDM

LINKITU         MACRO
                LINKLIB _LVO\1,_IntuitionBase
                ENDM
```

HP_Laserjet: printertag.asm

```
**********************************************************************
*
*       printer device dependent code tag
*
**********************************************************************

        SECTION         printer

*------ Included Files -----------------------------------------------

        INCLUDE         "exec/types.i"
        INCLUDE         "exec/nodes.i"
        INCLUDE         "exec/strings.i"

        INCLUDE         "hp_rev.i"

        INCLUDE         "devices/prtbase.i"


*------ Imported Names -----------------------------------------------

        XREF            _Init
        XREF            _Expunge
        XREF            _Open
        XREF            _Close
        XREF            _CommandTable
        XREF            _PrinterSegmentData
        XREF            _DoSpecial
        XREF            _Render
        XREF            _ExtendedCharTable
        XREF            _ConvFunc

*------ Exported Names --------------------------------------------

        XDEF            _PEDData

**********************************************************************

                MOVEQ   #0,D0           ; show error for OpenLibrary()
                RTS
                DC.W    VERSION
                DC.W    REVISION
_PEDData:
                DC.L    printerName
                DC.L    _Init
                DC.L    _Expunge
                DC.L    _Open
                DC.L    _Close
                DC.B    PPC_BWGFX       ; PrinterClass
                DC.B    PCC_BW          ; ColorClass
                DC.B    0               ; MaxColumns
                DC.B    0               ; NumCharSets
                DC.W    1               ; NumRows
                DC.L    600             ; MaxXDots
                DC.L    795             ; MaxYDots
                DC.W    75              ; XDotsInch
                DC.W    75              ; YDotsInch
                DC.L    _CommandTable   ; Commands
                DC.L    _DoSpecial
                DC.L    _Render
                DC.L    30              ; Timeout
                DC.L    _ExtendedCharTable      ; 8BitChars
                DS.L    1               ; PrintMode (reserve space)
                DC.L    _ConvFunc       ; ptr to char conversion function

printerName:
                dc.b    'HP_LaserJet',0

                END
```

HP_Laserjet: hp_rev.i

```
VERSION         EQU     35
REVISION        EQU     1
```

HP_Laserjet: init.asm

```
**********************************************************************
*
*       printer device functions
*
**********************************************************************

        SECTION         printer

*------ Included Files --------------------------------------------

        INCLUDE         "exec/types.i"
        INCLUDE         "exec/nodes.i"
        INCLUDE         "exec/lists.i"
        INCLUDE         "exec/memory.i"
        INCLUDE         "exec/ports.i"
        INCLUDE         "exec/libraries.i"

        INCLUDE         "macros.i"
```

```
*------ Imported Functions ------------------------------------------

        XREF_EXE        CloseLibrary
        XREF_EXE        OpenLibrary
        XREF            _AbsExecBase

        XREF            _PEDData

*------ Exported Globals --------------------------------------------

        XDEF            _Init
        XDEF            _Expunge
        XDEF            _Open
        XDEF            _PD
        XDEF            _PED
        XDEF            _SysBase
        XDEF            _DOSBase
        XDEF            _GfxBase
        XDEF            _IntuitionBase

**********************************************************************
        SECTION         printer,DATA
_PD             DC.L    0
_PED            DC.L    0
_SysBase        DC.L    0
_DOSBase        DC.L    0
_GfxBase        DC.L    0
_IntuitionBase  DC.L    0

**********************************************************************
        SECTION         printer,CODE
_Init:
                MOVE.L  4(A7),_PD
                LEA     _PEDData(PC),A0
                MOVE.L  A0,_PED
                MOVE.L  A6,-(A7)
                MOVE.L  _AbsExecBase,A6
                MOVE.L  A6,_SysBase

*           ;------ open the dos library
                LEA     DLName(PC),A1
                MOVEQ   #0,D0
                CALLEXE OpenLibrary
                MOVE.L  D0,_DOSBase
                BEQ     initDLErr

*           ;------ open the graphics library
                LEA     GLName(PC),A1
                MOVEQ   #0,D0
                CALLEXE OpenLibrary
                MOVE.L  D0,_GfxBase
                BEQ     initGLErr

*           ;------ open the intuition library
                LEA     ILName(PC),A1
                MOVEQ   #0,D0
                CALLEXE OpenLibrary
                MOVE.L  D0,_IntuitionBase
                BEQ     initILErr

                MOVEQ   #0,D0
pdiRts:
                MOVE.L  (A7)+,A6
                RTS

initPAErr:
                MOVE.L  _IntuitionBase,A1
                LINKEXE CloseLibrary
initILErr:
                MOVE.L  _GfxBase,A1
                LINKEXE CloseLibrary
initGLErr:
                MOVE.L  _DOSBase,A1
                LINKEXE CloseLibrary
initDLErr:
                MOVEQ   #-1,D0
                BRA.S   pdiRts
```

```
ILName:
                DC.B    'intuition.library'
                DC.B    0
DLName:
                DC.B    'dos.library'
                DC.B    0
GLName:
                DC.B    'graphics.library'
                DC.B    0
                DS.W    0

*----------------------------------------------------------------------
_Expunge:
                MOVE.L  _IntuitionBase,A1
                LINKEXE CloseLibrary

                MOVE.L  _GfxBase,A1
                LINKEXE CloseLibrary

                MOVE.L  _DOSBase,A1
                LINKEXE CloseLibrary

*----------------------------------------------------------------------
_Open:
                MOVEQ   #0,D0
                RTS

                END
```

HP_Laserjet: data.c

```
/*
        Data.c table for HP_LaserJet (Plus and II compatible) driver.
*/

char *CommandTable[] = {
        "\375\033E\375",/* 00 aRIS reset                        */
        "\377",         /* 01 aRIN initialize                   */
        "\012",         /* 02 aIND linefeed                     */
        "\015\012",     /* 03 aNEL CRLF                         */
        "\033&a-1R",    /* 04 aRI reverse LF                    */

                        /* 05 aSGR0 normal char set             */
        "\033&d@\033(sbS",
        "\033(s1S",     /* 06 aSGR3 italics on                  */
        "\033(sS",      /* 07 aSGR23 italics off                */
        "\033&dD",      /* 08 aSGR4 underline on                */
        "\033&d@",      /* 09 aSGR24 underline off              */
        "\033(s5B",     /* 10 aSGR1 boldface on                 */
        "\033(sB",      /* 11 aSGR22 boldface off               */
        "\377",         /* 12 aSFC set foreground color         */
        "\377",         /* 13 aSBC set background color         */

        "\033(s10h1T",  /* 14 aSHORP0 normal pitch              */
        "\033(s12h2T",  /* 15 aSHORP2 elite on                  */
        "\033(s10h1T",  /* 16 aSHORP1 elite off                 */
        "\033(s15H",    /* 17 aSHORP4 condensed fine on         */
        "\033(s10H",    /* 18 aSHORP3 condensed fine off        */
        "\377",         /* 19 aSHORP6 enlarge on                */
        "\377",         /* 20 aSHORP5 enlarge off               */

        "\033(s7B",     /* 21 aDEN6 shadow print on             */
        "\033(sB",      /* 22 aDEN5 shadow print off            */
        "\033(s3B",     /* 23 aDEN4 double strike on            */
        "\033(sB",      /* 24 aDEN3 double strike off           */
        "\377",         /* 25 aDEN2 NLQ on                      */
        "\377",         /* 26 aDEN1 NLQ off                     */

        "\377",         /* 27 aSUS2 superscript on              */
        "\377",         /* 28 aSUS1 superscript off             */
        "\377",         /* 29 aSUS4 subscript on                */
        "\377",         /* 30 aSUS3 subscript off               */
        "\377",         /* 31 aSUS0 normalize the line          */
        "\033&a-.5R",   /* 32 aPLU partial line up              */
        "\033=",        /* 33 aPLD partial line down          */

        "\033(s3T",     /* 34 aFNT0 Typeface 0                */
        "\033(s0T",     /* 35 aFNT1 Typeface 1                */
        "\033(s1T",     /* 36 aFNT2 Typeface 2                */
        "\033(s2T",     /* 37 aFNT3 Typeface 3                */
        "\033(s4T",     /* 38 aFNT4 Typeface 4                */
        "\033(s5T",     /* 39 aFNT5 Typeface 5                */
        "\033(s6T",     /* 40 aFNT6 Typeface 6                */
        "\033(s7T",     /* 41 aFNT7 Typeface 7                */
        "\033(s8T",     /* 42 aFNT8 Typeface 8                */
        "\033(s9T",     /* 43 aFNT9 Typeface 9                */
        "\033(s10T",    /* 44 aFNT10 Typeface 10              */

        "\033(s1P",     /* 45 aPROP2 proportional on          */
        "\033(sP",      /* 46 aPROP1 proportional off         */
        "\033(sP",      /* 47 aPROP0 proportional clear       */
        "\377",         /* 48 aTSS set proportional offset    */
        "\377",         /* 49 aJFY5 auto left justify         */
        "\377",         /* 50 aJFY7 auto right justify        */
        "\377",         /* 51 aJFY6 auto full jusitfy         */
        "\377",         /* 52 aJFY0 auto jusity off           */
        "\377",         /* 53 aJFY3 letter space              */
        "\377",         /* 54 aJFY1 word fill                 */

        "\033&l8D",     /* 55 aVERP0 1/8" line spacing        */
        "\033&l6D",     /* 56 aVERP1 1/6" line spacing        */
        "\377",         /* 57 aSLPP set form length           */
        "\033&l1L",     /* 58 aPERF perf skip n (n > 0)        */
        "\033&lL",      /* 59 aPERF0 perf skip off            */

        "\377",         /* 60 aLMS set left margin            */
        "\377",         /* 61 aRMS set right margin           */
        "\377",         /* 62 aTMS set top margin             */
        "\377",         /* 63 aBMS set bottom margin          */
        "\377",         /* 64 aSTBM set T&B margins           */
        "\377",         /* 65 aSLRM set L&R margins           */
        "\0339\015",    /* 66 aCAM clear margins              */

        "\377",         /* 67 aHTS set horiz tab              */
        "\377",         /* 68 aVTS set vert tab               */
        "\377",         /* 69 aTBC0 clear horiz tab           */
        "\377",         /* 70 aTBC3 clear all horiz tabs      */
        "\377",         /* 71 aTBC1 clear vert tab            */
        "\377",         /* 72 aTBC4 clear all vert tabs       */
        "\377",         /* 73 aTBCALL clear all h & v tabs    */
        "\377",         /* 74 aTBSALL set default tabs        */

        "\377",         /* 75 aEXTEND extended commands       */
        "\377"          /* 76 aRAW next 'n' chars are raw     */
};

char *ExtendedCharTable[] = {
/*
    " ", "!", "c", "L", "o", "Y", "|", "S",

    "\"", "c", "a", "<", "-", "-", "r", "-",

    "*", "+", "2", "3", "'", "u", "P", ".",

    ",", "1", "o", ">", "/", "/", "/", "?",

    "A", "A", "A", "A", "A", "A", "A", "C",

    "E", "E", "E", "E", "I", "I", "I", "I",

    "D", "N", "O", "O", "O", "O", "O", "x",

    "O", "U", "U", "U", "U", "Y", "P", "B",

    "a", "a", "a", "a", "a", "a", "a", "c",

    "e", "e", "e", "e", "i", "i", "i", "i",

    "d", "n", "o", "o", "o", "o", "o", "/",

    "o", "u", "u", "u", "u", "y", "p", "y"
*/

        " ", "\270", "\277", "\273", "\272", "\274", "|", "\275",
        "\253", "c", "\371", "\373", "~", "\366", "r", "\260",
        "\263", "\376", "2", "3", "\250", "\363", "\364", "\362",
        ",", "1", "\372", "\375", "\367", "\370", "\365", "\271",
        "\241", "\340", "\242", "\341", "\330", "\320", "\323", "\264",
        "\243", "\334", "\244", "\245", "\346", "\345", "\246", "\247",
        "\343", "\266", "\350", "\347", "\337", "\351", "\332", "x",
        "\322", "\255", "\355", "\256", "\333", "\261", "\360", "\336",
        "\310", "\304", "\300", "\342", "\314", "\324", "\327", "\265",
        "\311", "\305", "\301", "\315", "\331", "\325", "\321", "\335",
        "\344", "\267", "\312", "\306", "\302", "\352", "\316", "-\010:",
        "\326", "\313", "\307", "\303", "\317", "\262", "\361", "\357"
};
```

HP_Laserjet: dospecial.c

```
/*
        DoSpecial for HP_LaserJet driver.
*/

#include "exec/types.h"
#include "devices/printer.h"
#include "devices/prtbase.h"

#define LPI             7
#define CPI             15
#define QUALITY         17
#define INIT_LEN        30
#define LPP             7
#define FORM_LEN        11
#define LEFT_MARG       3
#define RIGHT_MARG      7
#define MARG_LEN        12

DoSpecial(command, outputBuffer, vline, currentVMI, crlfFlag, Parms)
char outputBuffer[];
UWORD *command;
BYTE *vline;
BYTE *currentVMI;
BYTE *crlfFlag;
UBYTE Parms[];
{
        extern struct PrinterData *PD;
        extern struct PrinterExtendedData *PED;

        static UWORD textlength, topmargin;
        int x, y, j;
        static char initThisPrinter[INIT_LEN] =
                "\033&d@\033&l6D\033(s0b10h1q0p0s3t0u12V";
        static char initForm[FORM_LEN] = "\033&l002e000F";
        static char initMarg[MARG_LEN] = "\033&a0001000M\015";
        static char initTMarg[] = "\033&l000e000F";

        x = y = j = 0;

        if (*command == aRIN) {
                while(x < INIT_LEN) {
                        outputBuffer[x] = initThisPrinter[x];
                        x++;
                }
                outputBuffer[x++] = '\015';

                if (PD->pd_Preferences.PrintSpacing == EIGHT_LPI) {
                        outputBuffer[LPI] = '8';
                }

                if (PD->pd_Preferences.PrintPitch == ELITE) {
                        outputBuffer[CPI] = '2';
                }
                else if (PD->pd_Preferences.PrintPitch == FINE) {
                        outputBuffer[CPI] = '5';
                }
        if (PD->pd_Preferences.PrintQuality == LETTER) {
                outputBuffer[QUALITY] = '2';
        }

        j = x; /* set the formlength = textlength, top margin = 2 */
        textlength = PD->pd_Preferences.PaperLength;
        topmargin = 2;

        while (y < FORM_LEN) {
                outputBuffer[x++] = initForm[y++];
        }
        numberString(textlength, j + LPP, outputBuffer);

        Parms[0] = PD->pd_Preferences.PrintLeftMargin;
        Parms[1] = PD->pd_Preferences.PrintRightMargin;
        *command = aSLRM;
}

if (*command == aSLRM) {
        j = x;
        y = 0;
        while(y < MARG_LEN) {
                outputBuffer[x++] = initMarg[y++];
        }
        numberString(Parms[0] - 1, j + LEFT_MARG, outputBuffer);
        numberString(Parms[1] - 1, j + RIGHT_MARG, outputBuffer);
        return(x);
}

if ((*command == aSUS2) && (*vline == 0)) {
        *command = aPLU;
        *vline = 1;
        return(0);
}

if ((*command == aSUS2) && (*vline < 0)) {
        *command = aRI;
        *vline = 1;
        return(0);
}

if ((*command == aSUS1) && (*vline > 0)) {
        *command = aPLD;
        *vline = 0;
        return(0);
}

if ((*command == aSUS4) && (*vline == 0)) {
        *command = aPLD;
        *vline = -1;
        return(0);
}

if ((*command == aSUS4) && (*vline > 0)) {
        *command = aIND;
        *vline = -1;
        return(0);
}

if ((*command == aSUS3) && (*vline < 0)) {
        *command = aPLU;
        *vline = 0;
        return(0);
}

if(*command == aSUS0) {
        if (*vline > 0) {
                *command = aPLD;
        }
        if (*vline < 0) {
                *command = aPLU;
        }
        *vline = 0;
        return(0);
}

if (*command == aPLU) {
```

```
                (*vline)++;
                return(0);
        }

        if (*command == aPLD){
                (*vline)--;
                return(0);
        }

        if (*command == aSTBM) {
                if (Parms[0] == 0) {
                        Parms[0] = topmargin;
                }
                else {
                        topmargin = --Parms[0];
                }

                if (Parms[1] == 0) {
                        Parms[1] = textlength;
                }
                else {
                        textlength=Parms[1];
                }
                while (x < 11) {
                        outputBuffer[x] = initTMarg[x];
                        x++;
                }
                numberString(Parms[0], 3, outputBuffer);
                numberString(Parms[1] - Parms[0], 7, outputBuffer);
                return(x);
        }

        if (*command == aSLPP) {
                while(x < 11) {
                        outputBuffer[x] = initForm[x];
                        x++;
                }
                /*restore textlength, margin*/
                numberString(topmargin, 3, outputBuffer);
                numberString(textlength, 7, outputBuffer);
                return(x);
        }

        if (*command == aRIS) {
                PD->pd_PWaitEnabled = 253;
        }

        return(0);
}

numberString(Param, x, outputBuffer)
UBYTE Param;
int x;
char outputBuffer[];
{
        if (Param > 199) {
                outputBuffer[x++] = '2';
                Param -= 200;
        }
        else if (Param > 99) {
                outputBuffer[x++] = '1';
                Param -= 100;
        }
        else {
                outputBuffer[x++] = '0'; /* always return 3 digits */
        }

        if (Param > 9) {
                outputBuffer[x++] = Param / 10 + '0';
        }
        else {
                outputBuffer[x++] = '0';
        }

        outputBuffer[x++] = Param % 10 + '0';
}
```

```
ConvFunc(buf, c, flag)
char *buf, c;
int flag; /* expand lf into lf/cr flag (0-yes, else no ) */
{
        if (c == '\014') { /* if formfeed (page eject) */
                PED->ped_PrintMode = 0; /* no data to print */
        }
        return(-1); /* pass all chars back to the printer device */
}

Close(ior)
struct printerIO *ior;
{
        if (PED->ped_PrintMode) { /* if data has been printed */
                (*(PD->pd_PWrite))("\014",1); /* eject page */
                (*(PD->pd_PBothReady))(); /* wait for it to finish */
                PED->ped_PrintMode = 0; /* no data to print */
        }
        return(0);
}
```

HP_Laserjet: render.c

```
/*
        HP_LaserJet driver.
*/

#include <exec/types.h>
#include <exec/nodes.h>
#include <exec/lists.h>
#include <exec/memory.h>
#include <devices/prtbase.h>
#include <devices/printer.h>

#define NUMSTARTCMD     7       /* # of cmd bytes before binary data */
#define NUMENDCMD       0       /* # of cmd bytes after binary data */
#define NUMTOTALCMD (NUMSTARTCMD + NUMENDCMD)   /* total of above */

extern SetDensity();
/*
        00-04   \033&10L        perf skip mode off
        05-11   \033*t075R      set raster graphics resolution (dpi)
        12-16   \033*r0A        start raster graphics
*/
char StartCmd[18] = "\033&10L\033*t075R\033*r0A";

Render(ct, x, y, status)
long ct, x, y, status;
{
        extern void *AllocMem(), FreeMem();

        extern struct PrinterData *PD;
        extern struct PrinterExtendedData *PED;

        static UWORD RowSize, BufSize, TotalBufSize, dataoffset;
        static UWORD huns, tens, ones; /* used to program buffer size */
        UBYTE *ptr, *ptrstart;
        int i, err;

        err=PDERR_NOERR;
        switch(status) {
                case 0 : /* Master Initialization */
                        /*
                                ct      - pointer to IODRPReq structure.
                                x       - width of printed picture in pixels.
                                y       - height of printed picture in pixels.
                        */
                        RowSize = (x + 7) / 8;
                        BufSize = RowSize + NUMTOTALCMD;
                        TotalBufSize = BufSize * 2;
                        PD->pd_PrintBuf = AllocMem(TotalBufSize, MEMF_PUBLIC);
                        if (PD->pd_PrintBuf == NULL) {
                                err = PDERR_BUFFERMEMORY; /* no mem */
                        }
                        else {
```

```
                ptr = PD->pd_PrintBuf;
                *ptr++ = 27;
                *ptr++ = '*';
                *ptr++ = 'b';   /* transfer raster graphics */
                *ptr++ = huns | '0';
                *ptr++ = tens | '0';
                *ptr++ = ones | '0';    /* printout width */
                *ptr = 'W';             /* terminator */
                ptr = &PD->pd_PrintBuf[BufSize];
                *ptr++ = 27;
                *ptr++ = '*';
                *ptr++ = 'b';   /* transfer raster graphics */
                *ptr++ = huns | '0';
                *ptr++ = tens | '0';
                *ptr++ = ones | '0';    /* printout width */
                *ptr = 'W';             /* terminator */
                dataoffset = NUMSTARTCMD;
        /* perf skip mode off, set dpi, start raster gfx */
                err = (*(PD->pd_PWrite))(StartCmd, 17);
        }
        break;

case 1 : /* Scale, Dither and Render */
        /*
                ct      - pointer to PrtInfo structure.
                x       - 0.
                y       - row # (0 to Height - 1).
        */
        Transfer(ct, y, &PD->pd_PrintBuf[dataoffset]);
        err = PDERR_NOERR; /* all ok */
        break;

case 2 : /* Dump Buffer to Printer */
        /*
                ct      - 0.
                x       - 0.
                y       - # of rows sent (1 to NumRows).
                White-space strip.
        */
        i = RowSize;
        ptrstart = &PD->pd_PrintBuf[dataoffset - NUMSTARTCMD];
        ptr = ptrstart + NUMSTARTCMD + i - 1;
        while (i > 0 && *ptr == 0) {
                i--;
                ptr--;
        }
        ptr = ptrstart + 3; /* get ptr to density info */
        *ptr++ = (huns = i / 100) | '0';
        *ptr++ = (i - huns * 100) / 10 | '0';
        *ptr = i % 10 | '0'; /* set printout width */
        err = (*(PD->pd_PWrite))(ptrstart, i + NUMTOTALCMD);
        if (err == PDERR_NOERR) {
                dataoffset = (dataoffset == NUMSTARTCMD ?
                        BufSize : 0) + NUMSTARTCMD;
        }
        break;

case 3 : /* Clear and Init Buffer */
        /*
                ct      - 0.
                x       - 0.
                y       - 0.
        */
        ptr = &PD->pd_PrintBuf[dataoffset];
        i = RowSize;
        do {
                *ptr++ = 0;
        } while (--i);
        break;

case 4 : /* Close Down */
        /*
                ct      - error code.
                x       - io_Special flag from IODRPReq struct
                y       - 0.
        */
        err = PDERR_NOERR; /* assume all ok */
```

```
                        /* if user did not cancel the print */
                        if (ct != PDERR_CANCEL) {
                                /* end raster graphics, perf skip mode on */
                                if ((err = (*(PD->pd_PWrite))
                                        ("\033*rB\033&l1L", 9)) == PDERR_NOERR) {
                                        /* if want to unload paper */
                                        if (!(x & SPECIAL_NOFORMFEED)) {
                                                /* eject paper */
                                                err = (*(PD->pd_PWrite))
                                                        ("\014", 1);
                                        }
                                }
                        }
                        /*
                                flag that there is no alpha data waiting that
                                needs a formfeed (since we just did one)
                        */
                        PED->ped_PrintMode = 0;
                         /* wait for both buffers to empty */
                        (*(PD->pd_PBothReady))();
                        if (PD->pd_PrintBuf != NULL) {
                                FreeMem(PD->pd_PrintBuf, TotalBufSize);
                        }
                        break;

                case 5 : /* Pre-Master Initialization */
                        /*
                                ct      - 0 or pointer to IODRPReq structure.
                                x       - io_Special flag from IODRPReq struct
                                y       - 0.
                        */
                        /* select density */
                        SetDensity(x & SPECIAL_DENSITYMASK);
                        break;
        }
        return(err);
}
```

HP_Laserjet: density.c

```
/*
        Density module for HP_LaserJet
*/

#include <exec/types.h>
#include <devices/printer.h>
#include <devices/prtbase.h>

SetDensity(density_code)
ULONG density_code;
{
        extern struct PrinterData *PD;
        extern struct PrinterExtendedData *PED;
        extern char StartCmd[];

        /* SPECIAL_DENSITY     0   1   2    3    4    5    6    7 */
        static int XDPI[8] = {75, 75, 100, 150, 300, 300, 300, 300};
        static char codes[8][3] = {
        {'0','7','5'},{'0','7','5'},{'1','0','0'},{'1','5','0'},
        {'3','0','0'},{'3','0','0'},{'3','0','0'},{'3','0','0'},
        };

        density_code /= SPECIAL_DENSITY1;
        PED->ped_MaxXDots = XDPI[density_code] * 8; /* 8 inches */

        /* default is 10.0, US_LEGAL is 14.0 */
        PED->ped_MaxYDots =
                PD->pd_Preferences.PaperSize == US_LEGAL ? 14 : 10;
        PED->ped_MaxYDots *= XDPI[density_code];

        PED->ped_XDotsInch = PED->ped_YDotsInch = XDPI[density_code];
        StartCmd[8] = codes[density_code][0];
        StartCmd[9] = codes[density_code][1];
        StartCmd[10] = codes[density_code][2];
}
```

HP_Laserjet transfer.c

```
/*
        Example transfer routine for HP_LaserJet driver.

        Transfer() should be written in assembly code for speed
*/

#include <exec/types.h>
#include <devices/prtgfx.h>

Transfer(PInfo, y, ptr)
struct PrtInfo *PInfo;
UWORD y;        /* row # */
UBYTE *ptr;     /* ptr to buffer */
{
        static UBYTE bit_table[] = {128, 64, 32, 16, 8, 4, 2, 1};
        UBYTE *dmatrix, Black, dvalue, threshold;
        union colorEntry *ColorInt;
        UWORD x, width, sx, *sxptr, bit;

        /* pre-compute */
        /* printer non-specific, MUST DO FOR EVERY PRINTER */
        x = PInfo->pi_xpos; /* get starting x position */
        ColorInt = PInfo->pi_ColorInt; /* get ptr to color intensities */
        sxptr = PInfo->pi_ScaleX;
        width = PInfo->pi_width; /* get # of source pixels */

        /* pre-compute threshold; are we thresholding? */
        if (threshold = PInfo->pi_threshold) { /* thresholding */
                dvalue = threshold ^ 15; /* yes, so pre-compute dither value */
                do { /* for all source pixels */
                        /* pre-compute intensity value for Black */
                        Black = ColorInt->colorByte[PCMBLACK];
                        ColorInt++; /* bump ptr for next time */

                        sx = *sxptr++;

                        /* dither and render pixel */
                        do { /* use this pixel 'sx' times */
                                /* if we should render Black */
                                if (Black > dvalue) {
                                        /* set bit */
                                        *(ptr + (x >> 3)) |= bit_table[x & 7];
                                }
                                ++x; /* done 1 more printer pixel */
                        } while (--sx);
                } while (--width);
        }
        else { /* not thresholding, pre-compute ptr to dither matrix */
                dmatrix = PInfo->pi_dmatrix + ((y & 3) << 2);
                do { /* for all source pixels */
                        /* pre-compute intensity value for Black */
                        Black = ColorInt->colorByte[PCMBLACK];
                        ColorInt++; /* bump ptr for next time */

                        sx = *sxptr++;

                        /* dither and render pixel */
                        do { /* use this pixel 'sx' times */
                                /* if we should render Black */
                                if (Black > dmatrix[x & 3]) {
                                        /* set bit */
                                        *(ptr + (x >> 3)) |= bit_table[x & 7];
                                }
                                ++x; /* done 1 more printer pixel */
                        } while (--sx);
                } while (--width);
        }
}
```

HP_Laserjet transfer.asm

```
**********************************************************************
*
* Transfer routine for HP_LaserJet
*
**********************************************************************

        INCLUDE "exec/types.i"

        INCLUDE "intuition/intuition.i"
        INCLUDE "devices/printer.i"
        INCLUDE "devices/prtbase.i"
        INCLUDE "devices/prtgfx.i"

        XREF    _PD

        XDEF    _Transfer

        SECTION         printer,CODE
_Transfer:
; Transfer(PInfo, y, ptr)
; struct PrtInfo *PInfo         4-7
; UWORD y;                      8-11
; UBYTE *ptr;                   12-15
;

        movem.l d2-d6/a2-a3,-(sp)       ;save regs used

        movea.l 32(sp),a0               ;a0 = PInfo
        move.l  36(sp),d0               ;d0 = y
        movea.l 40(sp),a1               ;a1 = ptr

        move.w  pi_width(a0),d1         ;d1 = width
        subq.w  #1,d1                   ;adjust for dbra

        move.w  pi_threshold(a0),d3     ;d3 = threshold, thresholding?
        beq.s   grey_scale              ;no, grey-scale

threshold:
; a0 - PInfo
; a1 - ptr
; d0 - y
; d1 - width
; d3 - threshold

        eori.b  #15,d3                  ;d3 = dvalue
        movea.l pi_ColorInt(a0),a2      ;a2 = ColorInt ptr
        move.w  pi_xpos(a0),d2          ;d2 = x
        movea.l pi_ScaleX(a0),a0        ;a0 = ScaleX (sxptr)

; a0 - sxptr
; a1 - ptr
; a2 - ColorInt ptr
; a3 - dmatrix ptr (NOT USED)
; d0 - byte to set (x >> 3)
; d1 - width
; d2 - x
; d3 - dvalue
; d4 - Black
; d5 - sx
; d6 - bit to set

twidth_loop:
        move.b  PCMBLACK(a2),d4         ;d4 = Black
        addq.l  #ce_SIZEOF,a2           ;advance to next entry

        move.w  (a0)+,d5                ;d5 = # of times to use this pixel (sx)

        cmp.b   d3,d4                   ;render this pixel?
        ble.s   tsx_end                 ;no, skip to next pixel.
        subq.w  #1,d5                   ;adjust for dbra

tsx_render:                             ;yes, render this pixel sx times
        move.w  d2,d0
        lsr.w   #3,d0                   ;compute byte to set
        move.w  d2,d6
        not.w   d6                      ;compute bit to set
        bset.b  d6,0(a1,d0.w)           ;*(ptr + x >> 3) |= 2 ^ x

        addq.w  #1,d2                   ;x++
        dbra    d5,tsx_render           ;sx--
        dbra    d1,twidth_loop          ;width--
        bra.s   exit                    ;all done

tsx_end:
        add.w   d5,d2                   ;x += sx
        dbra    d1,twidth_loop          ;width--
        bra.s   exit

grey_scale:
; a0 - PInfo
; a1 - ptr
; d0 - y
; d1 - width

        movea.l pi_ColorInt(a0),a2      ;a2 = ColorInt ptr
        moveq.l #3,d2
        and.w   d0,d2                   ;d2 = y & 3
        lsl.w   #2,d2                   ;d2 = (y & 3) << 2
        movea.l pi_dmatrix(a0),a3       ;a3 = dmatrix
        adda.l  d2,a3                   ;a3 = dmatrix + ((y & 3) << 2)
        move.w  pi_xpos(a0),d2          ;d2 = x
        movea.l pi_ScaleX(a0),a0        ;a0 = ScaleX (sxptr)

; a0 - sxptr
; a1 - ptr
; a2 - ColorInt ptr
; a3 - dmatrix ptr
; d0 - byte to set (x >> 3)
; d1 - width
; d2 - x
; d3 - dvalue (dmatrix[x & 3])
; d4 - Black
; d5 - sx
; d6 - bit to set

gwidth_loop:
        move.b  PCMBLACK(a2),d4         ;d4 = Black
        addq.l  #ce_SIZEOF,a2           ;advance to next entry

        move.w  (a0)+,d5                ;d5 = # of times to use this pixel (sx)
        subq.w  #1,d5                   ;adjust for dbra

gsx_loop:
        moveq.l #3,d3
        and.w   d2,d3                   ;d3 = x & 3
        move.b  0(a3,d3.w),d3           ;d3 = dmatrix[x & 3]

        cmp.b   d3,d4                   ;render this pixel?
        ble.s   gsx_end                 ;no, skip to next pixel.

        move.w  d2,d0
        lsr.w   #3,d0                   ;compute byte to set
        move.w  d2,d6
        not.w   d6                      ;compute bit to set
        bset.b  d6,0(a1,d0.w)           ;*(ptr + x >> 3) |= 2 ^ x

gsx_end
        addq.w  #1,d2                   ;x++
        dbra    d5,gsx_loop             ;sx--
        dbra    d1,gwidth_loop          ;width--

exit:
        movem.l (sp)+,d2-d6/a2-a3       ;restore regs used
        moveq.l #0,d0                   ;flag all ok
        rts                             ;goodbye

        END
```

Additional Information on the Printer Device

Additional programming information on the printer device can be found in the include files and the Autodocs for the printer device. Both are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

Printer Device Information	
INCLUDES	devices/printer.h
	devices/printer.i
	devices/prtbase.h
	devices/prtbase.i
	devices/prtgfx.h
	devices/prtgfx.i
AUTODOCS	printer.doc

Additional printer drivers can be found on Fred Fish Disk #344 under RKMCompanion.

chapter eleven
SCSI DEVICE

The Small Computer System Interface (SCSI) hardware of the A3000 and A2091/A590 is controlled by the SCSI device. The SCSI device allows an application to send Exec I/O commands and SCSI commands to a SCSI peripheral. Common SCSI peripherals include hard drives, streaming tape units and CD-ROM drives.

SCSI Device Commands and Functions

SCSI Device Command	**Operation**
HD_SCSICMD	Issue a SCSI-direct command to a SCSI unit.

Trackdisk Device Commands Supported by the SCSI Device

TD_CHANGESTATE	Return the disk present/not-present status of a drive.
TD_FORMAT	Initialize one or more tracks with a data buffer.
TD_PROTSTATUS	Return the write-protect status of a disk.
TD_SEEK	Move the head to a specific track.

Exec Commands Supported by SCSI Device

CMD_READ	Read one or more sectors from a disk.
CMD_START	Restart a SCSI unit that was previously stopped with CMD_STOP.
CMD_STOP	Stop a SCSI unit.
CMD_WRITE	Write one or more sectors to a disk.

Exec Functions as Used in This Chapter

AbortIO()	Abort an I/O request to the SCSI device.
AllocMem()	Allocate a block of memory.
AllocSignal()	Allocate a signal bit.
CheckIO()	Return the status of an I/O request.
CloseDevice()	Relinquish use of the SCSI device. All requests must be complete before closing.
DoIO()	Initiate a command and wait for completion (synchronous request).
FreeMem()	Free a block of previously allocated memory.
FreeSignal()	Free a previously allocated signal.
OpenDevice()	Obtain use of the SCSI device. You specify the type of unit and its characteristics in the call to **OpenDevice()**.
WaitIO()	Wait for completion of an I/O request and remove it from the reply port.

Exec Support Functions as Used in This Chapter

CreateExtIO	Create an extended **IORequest** structure for use in communicating with the SCSI device.
CreatePort()	Create a message port for reply messages from the SCSI device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete the extended **IORequest** structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The SCSI device operates like other Amiga devices. To use it, you must first open the SCSI device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The power of the SCSI device comes from its special facility for passing SCSI and SCSI-2 command blocks to any SCSI unit on the bus. This facility is commonly called *SCSI-direct* and it allows the Amiga to perform SCSI functions that are "non-standard" in terms of the normal Amiga I/O model.

To send SCSI-direct or other commands to the SCSI device, an extended I/O request structure named **IOStdReq** is used.

```
struct IOStdReq
{
    struct  Message io_Message;
    struct  Device  *io_Device;     /* device node pointer  */
    struct  Unit    *io_Unit;       /* unit (driver private)*/
    UWORD   io_Command;             /* device command */
    UBYTE   io_Flags;
    BYTE    io_Error;               /* error or warning num */
    ULONG   io_Actual;              /* actual number of bytes transferred */
    ULONG   io_Length;              /* requested number bytes transferred*/
    APTR    io_Data;                /* points to data area */
    ULONG   io_Offset;              /* offset for block structured devices */
};
```

See the include file *exec/io.h* for the complete structure definition.

OPENING THE SCSI DEVICE

Three primary steps are required to open the SCSI device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an I/O request structure of type **IOStdReq**. The **IOStdReq** structure is created by the **CreateExtIO()** function. **CreateExtIO** will initialize your **IOStdReq** to point to your reply port.
- Open the SCSI device. Call **OpenDevice()** passing it the I/O request and the SCSI unit encoded in the **unit** field.

SCSI unit encoding consists of three decimal digits which refer to the SCSI Target ID (bus address) in the 1s digit, the SCSI logical unit (LUN) in the 10s digit, and the controller board in the 100s digit. For example:

SCSI unit	Meaning
6	drive at address 6
12	LUN 1 on multiple drive controller at address 2
104	second controller board, address 4
88	not valid: both logical units and addresses range from 0–7

The Commodore 2090/2090A/2091 unit numbers are encoded differently. The SCSI logical unit (LUN) is in the 100s digit, and the SCSI Target ID is a permuted 1s digit: Target ID 0–6 maps to unit 3–9 (7 is reserved for the controller).

2090/A/1 unit	**Meaning**
3	drive at address 0
109	drive at address 6, logical unit 1
1	not valid: this is not a SCSI unit. Perhaps it's an ST506 unit.

Some controller boards generate a unique name for the second controller board, instead of implementing the 100s digit (e.g., the 2090A's iddisk.device).

```
struct MsgPort *SCSIMP;       /* Message port pointer */
struct IOStdReq *SCSIIO;      /* IORequest pointer */

    /* Create message port */
if (!(SCSIMP = CreatePort(NULL,NULL)))
    cleanexit("Can't create message port\n",RETURN_FAIL);

    /* Create IORequest */
if (!(SCSIIO = CreateExtIO(SCSIMP,sizeof(struct IOStdReq))))
    cleanexit("Can't create IORequest\n",RETURN_FAIL);

    /* Open the SCSI device */
if (error = OpenDevice("scsi.device",6L,SCSIIO,0L))
    cleanexit("Can't open scsi.device\n",RETURN_FAIL);
```

In the code above, the SCSI unit at address 6 of logical unit 0 of board 0 is opened.

CLOSING THE SCSI DEVICE

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**. All I/O requests must be complete before calling **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()**.

```
if (!(CheckIO(SCSIIO)))
    {
    AbortIO(SCSIIO);       /* Ask device to abort any pending requests */
    }
WaitIO(SCSIIO);            /* Wait for abort, then clean up */
CloseDevice(SCSIIO);       /* Close SCSI device */
```

SCSI-Direct

SCSI-direct is the facility of the Amiga's SCSI device interface that allows low-level SCSI commands to be passed directly to a SCSI unit on the bus. This makes it possible to support the special features of tape drives, hard disks and other SCSI equipment that do not fit into the Amiga's normal I/O model. For example, with SCSI-direct, special commands can be sent to hard drives to modify various drive parameters that are normally inaccessible or which differ from drive to drive.

In order to use SCSI-direct, you must first open the SCSI device for the unit you want to use in the manner described above. You then send an HD_SCSICMD I/O request with a pointer to a SCSI command data structure.

The SCSI device uses a special data structure for SCSI-direct commands named **SCSICmd**.

```
struct SCSICmd
{
    UWORD  *scsi_Data;          /* word aligned data for SCSI Data Phase */
                                /* (optional) data need not be byte aligned */
                                /* (optional) data need not be bus accessible */
    ULONG   scsi_Length;        /* even length of Data area */
                                /* (optional) data can have odd length */
                                /* (optional) data length can be > 2**24 */
    ULONG   scsi_Actual;        /* actual Data used */
    UBYTE  *scsi_Command;       /* SCSI Command (same options as scsi_Data) */
    UWORD   scsi_CmdLength;     /* length of Command */
    UWORD   scsi_CmdActual;     /* actual Command used */
    UBYTE   scsi_Flags;         /* includes intended data direction */
    UBYTE   scsi_Status;        /* SCSI status of command */
    UBYTE  *scsi_SenseData;     /* sense data: filled if SCSIF_[OLD]AUTOSENSE */
                                /* is set and scsi_Status has CHECK CONDITION */
                                /* (bit 1) set */
    UWORD   scsi_SenseLength;   /* size of scsi_SenseData, also bytes to */
                                /* request w/ SCSIF_AUTOSENSE, must be 4..255 */
    UWORD   scsi_SenseActual;   /* amount actually fetched (0 means no sense) */
};
```

See the include file *devices/scsidisk.h* for the complete structure definition.

SCSICmd will contain the SCSI command and any associated data that you wish to pass to the SCSI unit. You set its fields to the values required by the unit and the command. When you have opened the SCSI device and set the **SCSICmd** to the proper values, you are ready for SCSI-direct.

You send a SCSI-direct command by passing an **IOStdReq** to the SCSI device with a pointer to the **SCSICmd** structure set in **io_Data**, the size of the **SCSICmd** structure set in **io_Length** and HD_SCSICMD set in **io_Command**.:

```
struct IOStdReq *SCSIreq = NULL;
struct SCSICmd Cmd;             /* where the actual SCSI command goes */

SCSIreq->io_Length  = sizeof(struct SCSICmd);
SCSIreq->io_Data    = (APTR)&Cmd;
SCSIreq->io_Command = HD_SCSICMD;
DoIO(SCSIreq);
```

The **SCSICmd** structure must be filled in prior to passing it to the SCSI unit. How it is filled in depends on the SCSI-direct being passed to the unit. Below is an example of setting up a **SCSICmd** structure for the MODE_SENSE SCSI-direct command.

```
UBYTE *buffer;                  /* a data buffer used for mode sense data */
UBYTE  Sense[20];               /* buffer for request sense data */
struct SCSICmd Cmd;             /* where the actual SCSI command goes */
static UBYTE ModeSense[]={ 0x1a,0,0xff,0,254,0 }; /* the command being sent */

Cmd.scsi_Data = (UWORD *)buffer;        /* where we put mode sense data */
Cmd.scsi_Length = 254;                  /* how much we will accept      */
Cmd.scsi_CmdLength = 6;                 /* length of the command        */
Cmd.scsi_Flags = SCSIF_AUTOSENSE |      /* do automatic REQUEST_SENSE   */
                 SCSIF_READ;            /* set expected data direction  */
Cmd.scsi_SenseData = (UBYTE *)Sense;    /* where sense data will go     */
Cmd.scsi_SenseLength = 18;              /* how much we will accept      */
Cmd.scsi_SenseActual = 0;               /* how much has been received   */

Cmd.scsi_Command=(UBYTE *)ModeSense;/* issuing a MODE_SENSE command     */
```

The fields of the **SCSICmd** are:

scsi_data
This field points to the data buffer for the SCSI data phase (if any is expected). It is generally the job of the driver software to ensure that the given buffer is DMA-accessible and to drop to programmed I/O if it isn't. The filing system provides a stop-gap fix for non-conforming drivers with the AddressMask parameter in *DEVS:mountlist*. For absolute safety, restrict all direct reads and writes to Chip RAM.

scsi_Length
This is the expected length of data to be transferred. If an unknown amount of data is to be transferred from target to host, set the **scsi_Length** to be larger than the maximum amount of data expected. Some controllers explicitly use **scsi_Length** as the amount of data to transfer. The A2091, A590 and A3000 drivers always do programmed I/O for data transfers under 256 bytes or when the DMA chip doesn't support the required alignment.

scsi_Actual
How much data was actually received from or sent to the SCSI unit in response to the SCSI-direct command.

scsi_Command
The SCSI-direct command.

scsi_CmdLength
The length of the SCSI-direct command in bytes.

scsi_CmdActual
The actual number of bytes of the SCSI-direct command that were transferred to the SCSI unit.

scsi_Flags
These flags contain the intended data direction for the SCSI command. It is not strictly necessary to set the data direction flag since the SCSI protocol will inform the driver which direction data transfers will be going. However, some controllers use this information to set up DMA before issuing the command . It can also be used as a sanity check in case the data phase goes the wrong way.

One flag in particular, is worth noting. SCSIF_AUTOSENSE is used to make the driver perform an automatic REQUEST SENSE if the target returns CHECK CONDITION for a SCSI command. The reason for having the driver do this is the multitasking nature of the Amiga. If two tasks were accessing the same drive and the first received a CHECK CONDITION, the second task would destroy the sense information when it sent a command. SCSIF_AUTOSENSE prevents the caller from having to make two I/O requests and removes this window of vulnerability.

scsi_Status
The status of the SCSI-direct command. The values returned in this field can be found in the SCSI specification. For example, 2 is CHECK_CONDITION.

scsi_SenseActual
If the SCSIF_AUTOSENSE flag is set, it is important to initialize this field to zero before issuing a SCSI command because some drivers don't support AUTOSENSE and won't initialize the field.

scsi_SenseData

This field is used only for SCSIF_AUTOSENSE. If a REQUEST SENSE command is directly sent to the driver, the data will be deposited in the buffer pointed to by **scsi_Data**.

Keep in mind that SCSI-direct is geared toward an initiator role so it can't be expected to perform target-like operations. You can only send commands to a device, not receive them from an initiator. There is no provision for SCSI messaging, either. This is due mainly to the interactive nature of the extended messages (such as synchronous transfer requests) which have to be handled by the driver because it knows the limitations of the controller card and has to be made aware of such protocol changes.

RigidDiskBlock – Fields and Implementation

The **RigidDiskBlock** (RDB) standard was borne out of the same development effort as HD_SCSICMD and as a result has a heavy bias towards SCSI. However, there is nothing in the RDB specification that makes it unusable for devices using other bus protocols. The XT style disks used in the A590 also support the RDB standard.

The RDB scheme was designed to allow the automatic mounting of all partitions on a hard drive and subsequent booting from the highest priority partition even if it has a soft loaded filing system. Disks can be removed from one controller and plugged into another (supporting the RDB scheme) and will carry with it all the necessary information for mounting and booting with them.

The preferred method of creating RigidDiskBlocks is with the *HDToolBox* program supplied by Commodore. Most controllers include an RDB editor or utility.

When a driver is initialized, it uses the information contained in the RDB to mount the required partitions and mark them as bootable if needed. The driver is also responsible for loading any filing systems that are required if they are not already available on the filesystem.resource list. Filesystems are added to the resource according to **DosType** and version number.

The following is a listing of *devices/hardblocks.h* that describes all the fields in the RDB specification.

```
/*--------------------------------------------------------------------
 *
 *      This file describes blocks of data that exist on a hard disk
 *      to describe that disk.  They are not generically accessable to
 *      the user as they do not appear on any DOS drive.  The blocks
 *      are tagged with a unique identifier, checksummed, and linked
 *      together.  The root of these blocks is the RigidDiskBlock.
 *
 *      The RigidDiskBlock must exist on the disk within the first
 *      RDB_LOCATION_LIMIT blocks.  This inhibits the use of the zero
 *      cylinder in an AmigaDOS partition: although it is strictly
 *      possible to store the RigidDiskBlock data in the reserved
 *      area of a partition, this practice is discouraged since the
 *      reserved blocks of a partition are overwritten by "Format",
 *      "Install", "DiskCopy", etc.  The recommended disk layout,
 *      then, is to use the first cylinder(s) to store all the drive
 *      data specified by these blocks: i.e. partition descriptions,
 *      file system load images, drive bad block maps, spare blocks,
 *      etc.
 *
 *      Though only 512 byte blocks are currently supported by the
 *      file system, this proposal tries to be forward-looking by
 *      making the block size explicit, and by using only the first
 *      256 bytes for all blocks but the LoadSeg data.
 *
 *--------------------------------------------------------------------*/
/*
 *  NOTE
 *      optional block addresses below contain $ffffffff to indicate
 *      a NULL address, as zero is a valid address
 */
struct RigidDiskBlock
{
    ULONG   rdb_ID;                 /* 4 character identifier */
    ULONG   rdb_SummedLongs;        /* size of this checksummed structure */
    LONG    rdb_ChkSum;             /* block checksum (longword sum to zero) */
    ULONG   rdb_HostID;             /* SCSI Target ID of host */
    ULONG   rdb_BlockBytes;         /* size of disk blocks */
    ULONG   rdb_Flags;              /* see below for defines */
    /* block list heads */
    ULONG   rdb_BadBlockList;       /* optional bad block list */
    ULONG   rdb_PartitionList;      /* optional first partition block */
    ULONG   rdb_FileSysHeaderList;  /* optional file system header block */
    ULONG   rdb_DriveInit;          /* optional drive-specific init code */
                                    /* DriveInit(lun,rdb,ior): "C" stk & d0/a0/a1 */
    ULONG   rdb_Reserved1[6];       /* set to $ffffffff */
    /* physical drive characteristics */
    ULONG   rdb_Cylinders;          /* number of drive cylinders */
    ULONG   rdb_Sectors;            /* sectors per track */
    ULONG   rdb_Heads;              /* number of drive heads */
    ULONG   rdb_Interleave;         /* interleave */
    ULONG   rdb_Park;               /* landing zone cylinder */
    ULONG   rdb_Reserved2[3];
    ULONG   rdb_WritePreComp;       /* starting cylinder: write precompensation */
    ULONG   rdb_ReducedWrite;       /* starting cylinder: reduced write current */
    ULONG   rdb_StepRate;           /* drive step rate */
    ULONG   rdb_Reserved3[5];
    /* logical drive characteristics */
    ULONG   rdb_RDBBlocksLo;        /* low block of range reserved for hardblocks */
    ULONG   rdb_RDBBlocksHi;        /* high block of range for these hardblocks */
    ULONG   rdb_LoCylinder;         /* low cylinder of partitionable disk area */
    ULONG   rdb_HiCylinder;         /* high cylinder of partitionable data area */
    ULONG   rdb_CylBlocks;          /* number of blocks available per cylinder */
    ULONG   rdb_AutoParkSeconds; /* zero for no auto park */
    ULONG   rdb_Reserved4[2];
    /* drive identification */
    char    rdb_DiskVendor[8];
    char    rdb_DiskProduct[16];
    char    rdb_DiskRevision[4];
    char    rdb_ControllerVendor[8];
    char    rdb_ControllerProduct[16];
    char    rdb_ControllerRevision[4];
    ULONG   rdb_Reserved5[10];
};

#define IDNAME_RIGIDDISK        0x5244534B      /* 'RDSK' */

#define RDB_LOCATION_LIMIT      16

#define RDBFB_LAST      0       /* no disks exist to be configured after */
#define RDBFF_LAST      0x01L   /*   this one on this controller */
#define RDBFB_LASTLUN   1       /* no LUNs exist to be configured greater */
#define RDBFF_LASTLUN   0x02L   /*   than this one at this SCSI Target ID */
#define RDBFB_LASTTID   2       /* no Target IDs exist to be configured */
#define RDBFF_LASTTID   0x04L   /*   greater than this one on this SCSI bus */
#define RDBFB_NORESELECT 3      /* don't bother trying to perform reselection */
#define RDBFF_NORESELECT 0x08L  /*   when talking to this drive */
#define RDBFB_DISKID    4       /* rdb_Disk... identification valid */
#define RDBFF_DISKID    0x10L
#define RDBFB_CTRLRID   5       /* rdb_Controller... identification valid */
#define RDBFF_CTRLRID   0x20L

/*--------------------------------------------------------------------*/
struct BadBlockEntry {
    ULONG   bbe_BadBlock;           /* block number of bad block */
    ULONG   bbe_GoodBlock;          /* block number of replacement block */
};
struct BadBlockBlock {
    ULONG   bbb_ID;                 /* 4 character identifier */
    ULONG   bbb_SummedLongs;        /* size of this checksummed structure */
    LONG    bbb_ChkSum;             /* block checksum (longword sum to zero) */
    ULONG   bbb_HostID;             /* SCSI Target ID of host */
    ULONG   bbb_Next;           /* block number of the next BadBlockBlock */
    ULONG   bbb_Reserved;
    struct BadBlockEntry bbb_BlockPairs[61]; /* bad block entry pairs */
    /* note [61] assumes 512 byte blocks */
};

#define IDNAME_BADBLOCK         0x42414442      /* 'BADB' */

/*------------------------------------------------------------------*/
struct PartitionBlock {
    ULONG   pb_ID;              /* 4 character identifier */
    ULONG   pb_SummedLongs;     /* size of this checksummed structure */
    LONG    pb_ChkSum;          /* block checksum (longword sum to zero) */
    ULONG   pb_HostID;          /* SCSI Target ID of host */
    ULONG   pb_Next;            /* block number of the next PartitionBlock */
    ULONG   pb_Flags;           /* see below for defines */
    ULONG   pb_Reserved1[2];
    ULONG   pb_DevFlags;        /* preferred flags for OpenDevice */
    UBYTE   pb_DriveName[32];   /* preferred DOS device name: BSTR form */
                                /* (not used if this name is in use) */
    ULONG   pb_Reserved2[15];   /* filler to 32 longwords */
    ULONG   pb_Environment[17]; /* environment vector for this partition */
    ULONG   pb_EReserved[15];   /* reserved for future environment vector */
};

#define IDNAME_PARTITION        0x50415254      /* 'PART' */

#define PBFB_BOOTABLE   0       /* this partition is intended to be bootable */
#define PBFF_BOOTABLE   1L      /*   (expected directories and files exist) */
#define PBFB_NOMOUNT    1       /* do not mount this partition (e.g. manually */
#define PBFF_NOMOUNT    2L      /*   mounted, but space reserved here) */

/*------------------------------------------------------------------*/
struct FileSysHeaderBlock {
    ULONG   fhb_ID;             /* 4 character identifier */
    ULONG   fhb_SummedLongs;    /* size of this checksummed structure */
    LONG    fhb_ChkSum;         /* block checksum (longword sum to zero) */
    ULONG   fhb_HostID;         /* SCSI Target ID of host */
    ULONG   fhb_Next;           /* block number of next FileSysHeaderBlock */
    ULONG   fhb_Flags;          /* see below for defines */
    ULONG   fhb_Reserved1[2];
    ULONG   fhb_DosType;        /* file system description: match this with */
                                /* partition environment's DE_DOSTYPE entry */
    ULONG   fhb_Version;        /* release version of this code */
    ULONG   fhb_PatchFlags;     /* bits set for those of the following that */
                                /*   need to be substituted into a standard */
                                /*   device node for this file system: e.g. */
                                /*   0x180 to substitute SegList & GlobalVec */
    ULONG   fhb_Type;           /* device node type: zero */
    ULONG   fhb_Task;           /* standard dos "task" field: zero */
    ULONG   fhb_Lock;           /* not used for devices: zero */
    ULONG   fhb_Handler;        /* filename to loadseg: zero placeholder */
    ULONG   fhb_StackSize;      /* stacksize to use when starting task */
    LONG    fhb_Priority;       /* task priority when starting task */
    LONG    fhb_Startup;        /* startup msg: zero placeholder */
    LONG    fhb_SegListBlocks;  /* first of linked list of LoadSegBlocks: */
                                /*   note that this entry requires some */
                                /*   processing before substitution */
    LONG    fhb_GlobalVec;      /* BCPL global vector when starting task */
    ULONG   fhb_Reserved2[23];  /* (those reserved by PatchFlags) */
    ULONG   fhb_Reserved3[21];
};

#define IDNAME_FILESYSHEADER    0x46534844      /* 'FSHD' */

/*------------------------------------------------------------------*/
struct LoadSegBlock {
    ULONG   lsb_ID;             /* 4 character identifier */
    ULONG   lsb_SummedLongs;    /* size of this checksummed structure */
    LONG    lsb_ChkSum;         /* block checksum (longword sum to zero) */
    ULONG   lsb_HostID;         /* SCSI Target ID of host */
    ULONG   lsb_Next;           /* block number of the next LoadSegBlock */
    ULONG   lsb_LoadData[123];  /* data for "loadseg" */
    /* note [123] assumes 512 byte blocks */
};

#define IDNAME_LOADSEG          0x4C534547      /* 'LSEG' */
```

HOW A DRIVER USES RDB

The information contained in the **RigidDiskBlock** and subsequent **PartitionBlocks**, et al., is used by a driver in the following manner.

After determining that the target device is a hard disk (using the SCSI-direct command INQUIRY), the driver will scan the first RDB_LOCATION_LIMIT (16) blocks looking for a block with the "RDSK" identifier and a correct sum-to-zero checksum. If no RDB is found then the driver will give up and not attempt to mount any partitions for this unit. If the RDB is found then the driver looks to see if there's a partition list for this unit (**rdb_PartitionList**). If none, then just the **rdb_Flags** will be used to determine if there are any LUNs or units after this one. This is used for early termination of the search for units on bootup.

If a partition list is present, and the partition blocks have the correct ID and checksum, then for each partition block the driver does the following.

1. Checks the PBFB_NOMOUNT flag. If set then this partition is just reserving space. Skip to the next partition without mounting the current one.

2. If PBFB_NOMOUNT is false, then the partition is to be mounted. The driver fetches the given drive name from **pb_DriveName**. This name will be of the form *dh0*, *work*, *wb_2.x* etc. A check is made to see if this name already exists on **eb_MountList** or DOS's device list. If it does, then the name is algorithmically altered to remove duplicates. The A590, A2091 and A3000 append *.n* (where *n* is a number) unless another name ending with *.n* is found. In that case the name is changed to *.n*+1 and the search for duplicates is retried.

3. Next the driver constructs a parameter packet for **MakeDosNode()** using the (possibly altered) drive name and information about the Exec device name and unit number. **MakeDosNode()** is called to create a DOS device node. **MakeDosNode()** constructs a filesystem startup message from the given information and fills in defaults for the ROM filing system.

4. If **MakeDosNode()** succeeds then the driver checks to see if the entry is using a standard ("DOS\0") filing system. If not then the routine for patching in non-standard filing systems is called (see "Alien File Systems" below).

5. Now that the DOS node has been set up and the correct filing system segment has been associated with it, the driver checks PBFB_BOOTABLE to see if this partition is marked as bootable. If the partition is not bootable, or this is not autoboot time (`DiagArea == 0`) then the driver simply calls **AddDosNode()** to enqueue the DOS device node. If the partition is bootable, then the driver constructs a boot node and enqueues it on **eb_MountList** using the boot priority from the environment vector. If this boot priority is -128 then the partition is not considered bootable.

ALIEN FILING SYSTEMS

When a filing system other than the ROM filing system is to be used, the following steps take place.

1. First, open filesystem.resource in preparation for finding the filesystem segment we want. If filesystem.resource doesn't exist then create it and add it via **AddResource**(). Under 2.0 the resource is created by the system early on in the initialization sequence. Under pre-V36 Kickstart, it is the responsibility of the first RDB driver to create it.
2. Scan filesystem.resource looking for a filesystem that matches the **DosType** and version that we want. If it exists go to step 4.
3. Since the driver couldn't find the filesystem it needed, it will have to load it from the RDB area. The list of **FileSysHeaderBlocks** (pointed to by the "RDSK" block) is scanned for a filesystem of the required **DosType** and version. If none is found then the driver will give up and abort the mounting of the partition. If the required filesystem is found, then it is **LoadSeg**()'ed from the "LSEG" blocks and added as a new entry to the filesystem.resource.
4. The **SegList** pointer of the found or loaded filesystem is held in the **FileSysEntry** structure (which is basically an environment vector for this filing system). Using the patch flags, the driver now patches the newly created environment vector (pointed to by the new **DosNode**) using the values in the **FileSysEntry** being used. This ensures that the partition will have the correct filing system set up with the correct mount variables using a shared **SegList**.

The **eb_Mountlist** will now be set up with prioritized bootnodes and maybe some non-bootable, but mounted partitions. The system bootstrap will now take over.

Amiga BootStrap

At priority -40 in the system module initialization sequence, after most other modules are initialized, appropriate expansion boards are configured. Appropriate boards will match a `FindConfigDev(, -1, -1)`—these are all boards on the expansion library board list. Furthermore, they will meet all of the following conditions:

1. CDB_CONFIGME set in **cd_Flags**,
2. ERTB_DIAGVALID set in **cd_Rom->er_Type**,
3. diagnostic area pointer (in **cd_Rom->er_Reserved0c**) is non-zero,
4. DAC_CONFIGTIME set in **da_Config**, and
5. at least one valid resident tag within the diagnostic area, the first of which is used by **InitResident**() below. This resident structure was patched to be valid during the ROM diagnostic routine run when the expansion library first initialized the board.

Boards meeting all these conditions are initialized with the standard **InitResident()** mechanism, with a NULL **SegList**. The board initialization code can find its **ConfigDev** structure with the expansion library's **GetCurrentBinding()** function. This is an appropriate time for drivers to **Enqueue()** a boot node on the expansion library's **eb_MountList** for use by the strap module below, and clear CDB_CONFIGME so a *C:BindDrivers* command will not try to initialize the board a second time.

This module will also enqueue nodes for 3.5" trackdisk device units. These nodes will be at the following priorities:

Priority	Drive
5	df0:
-10	df1:
-20	df2:
-30	df3:

Next, at priority -60 in the system module initialization sequence, the strap module is invoked. Nodes from the prioritized **eb_MountList** list is used in priority order in attempts to boot. An item on the list is given a chance to boot via one of two different mechanisms, depending on whether it it uses boot code read in off the disk (*BootBlock booting*), or uses boot code provided in the device **ConfigDev** diagnostic area (*BootPoint booting*). Floppies always use the BootBlock method. Other entries put on the **eb_MountList** (e.g. hard disk partitions) used the BootPoint mechanism for pre-V36 Kickstart, but can use either for V36/V37.

The **eb_MountList** is modified before each boot attempt, and then restored and re-modified for the next attempt if the boot fails:

1. The node associated with the current boot attempt is placed at the head of the **eb_MountList**.
2. Nodes marked as unusable under AmigaDOS are removed from the list. Nodes that are unusable are marked by the longword **bn_DeviceNode->dn_Handler** having the most significant bit set. This is used, for example, to keep UNIX partitions off the AmigaDOS device list when booting AmigaDOS instead of UNIX.

The selection of which of the two different boot mechanisms to use proceeds as follows:

1. The node must be valid boot node, i.e. meet both of the following conditions:
   a) **ln_Type** is NT_BOOTNODE,
   b) **bn_DeviceNode** is non-zero,
2. The type of boot is determined by looking at the **DosEnvec** pointed to by **fssm_Environ** pointed to by the **dn_Startup** in the **bn_DeviceNode**:
   a) if the **de_TableSize** is less than DE_BOOTBLOCKS, or the **de_BootBlocks** entry is zero, BootPoint booting is specified, otherwise
   b) **de_BootBlocks** contains the number of blocks to read in from the beginning of the partition, checksum, and try to boot from.

BOOTBLOCK BOOTING

In BootBlock booting the sequence of events is as follows:

1. The disk device must contain valid boot blocks:
   a) the device and unit from **dn_Startup** opens successfully,
   b) memory is available for the <**de_BootBlocks**> * <**de_SizeBlock**> * 4 bytes of boot block code,
   c) the device commands CMD_CLEAR, TD_CHANGENUM, and CMD_READ of the boot blocks execute without error,
   d) the boot blocks start with the three characters "DOS" and pass the longword checksum (with carry wraparound), and
   e) memory is available to construct a boot node on the **eb_MountList** to describe the floppy.
   If a device error is reported in 1.c., or if memory is not available for 1.b. or 1.e., a recoverable alert is presented before continuing.
2. The boot code in the boot blocks is invoked as follows:
   a) The address of the entry point for the boot code is offset BB_ENTRY into the boot blocks in memory.
   b) The boot code is invoked with the I/O request used to issue the device commands in 1.c. above in register A1, with the **io_Offset** pointing to the beginning of the partition (the origin of the boot blocks) and **SysBase** in A6.
3. The boot code returns with results in both D0 and A0.
   a) Non-zero D0 indicates boot failure. The recoverable alert **AN_BootError** is presented before continuing.
   b) Zero D0 indicates A0 contains a pointer to the function to complete the boot. This completion function is chained to with **SysBase** in A6 after the strap module frees all its resources. It is usually the dos.library initialization function, from the dos.library resident tag. Return from this function is identical to return from the strap module itself.

BOOTPOINT BOOTING

BootPoint booting follows this sequence:

1. The **eb_MountList** node must contain a valid BootPoint:
   a) **ConfigDev** pointer (in **ln_Name**) is non-zero,
   b) diagnostic area pointer (in **cd_Rom er_Reserved0c**) is non-zero,
   c) DAC_CONFIGTIME set in **da_Config**.
2. The boot routine of a valid boot node is invoked as follows:
   a) The address of the boot routine is calculated from **da_BootPoint**.
   b) The resulting boot routine is invoked with the **ConfigDev** pointer on the stack in C fashion (i.e., `(*boot)(configDev);`). Moreover, register A2 will contain the address of the associated **eb_MountList** node.
3. Return from the boot routine indicates failure to boot.

If all entries fail to boot, the user is prompted to put a bootable disk into a floppy drive with the "strap screen". The system floppy drives are polled for new disks. When one appears, the "strap screen" is removed and the appropriate boot mechanism is applied as described above. The process of prompting and trying continues till a successful boot occurs.

SCSI-Direct Example

```
/*
 * SCSI_Direct.c
 *
 * The following program demonstrates the use of the HD_SCSICmd to send a
 * MODE SENSE to a unit on the requested device (default scsi.device).  This
 * code can be easily modified to send other commands to the drive.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/scsidisk.h>
#include <dos/dosextens.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdlib.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

#define BUFSIZE 256

UBYTE *buffer;                  /* a data buffer used for mode sense data */
struct IOStdReq SCSIReq;        /* a standard IORequest structure */
struct SCSICmd Cmd;             /* where the actual SCSI command goes */
UBYTE  Sense[20];               /* buffer for request sense data */
struct MsgPort Port;            /* our ReplyPort */

void ShowSenseData(void);

static UBYTE TestReady[] = { 0,0,0,0,0,0 };     /* not used but here for  */
static UBYTE StartUnit[] = { 0x1b,0,0,0,1,0 };  /* illustration of other  */
static UBYTE StopUnit[] =  { 0x1b,0,0,0,0,0 };  /* commands.              */

static UBYTE ModeSense[]={ 0x1a,0,0xff,0,254,0 }; /* the command being sent */

void main(int argc, char **argv)
{
int unit,tval,i;
char *dname = "scsi.device";
UBYTE *tbuf;

if ((argc < 2) || (argc > 3))
    {
    printf("Usage: %s unit [xxxx.device]\n",argv[0]);
    exit(100);
    }

unit = atoi( argv[1] );
if (argc == 3)
    dname = argv[2];

buffer = (UBYTE *) AllocMem(BUFSIZE, MEMF_PUBLIC|MEMF_CLEAR);

if (!buffer)
    {
    printf("Couldn't get memory\n");
    exit(100);
    }

Port.mp_Node.ln_Pri = 0;                        /* setup the ReplyPort */
Port.mp_SigBit      = AllocSignal(-1);
Port.mp_SigTask     = (struct Task *)FindTask(0);
NewList( &(Port.mp_MsgList) );
```

```
SCSIReq.io_Message.mn_ReplyPort = &Port;

if (OpenDevice( dname, unit, &SCSIReq, 0))
    {
    printf("Couldn't open unit %ld on %s\n",unit,dname);
    FreeMem( buffer,BUFSIZE );
    exit(100);
    }

SCSIReq.io_Length  = sizeof(struct SCSICmd);
SCSIReq.io_Data    = (APTR)&Cmd;
SCSIReq.io_Command = HD_SCSICMD;        /* the command we are sending   */

Cmd.scsi_Data = (UWORD *)buffer;        /* where we put mode sense data */
Cmd.scsi_Length = 254;          /* how much we will accept      */
Cmd.scsi_CmdLength = 6;         /* length of the command        */
Cmd.scsi_Flags = SCSIF_AUTOSENSE|SCSIF_READ;
                                /* do automatic REQUEST_SENSE   */
                                /* set expected data direction  */
Cmd.scsi_SenseData =(UBYTE *)Sense;     /* where sense data will go     */
Cmd.scsi_SenseLength = 18;              /* how much we will accept      */
Cmd.scsi_SenseActual = 0;               /* how much has been received   */

Cmd.scsi_Command=(UBYTE *)ModeSense;/* issuing a MODE_SENSE command     */
DoIO( &SCSIReq );                       /* send it to the device driver */

if (Cmd.scsi_Status)
    ShowSenseData();      /* if bad status then show it */

else
    {
    printf("\nBlock descriptor header\n");
    printf("=======================\n");
    printf("Mode Sense data length  = %d\n",(short)buffer[0]);
    printf("Block descriptor length = %d\n",(short)buffer[3]);
    tbuf = &buffer[4];
    printf("Density code            = %d\n",(short)tbuf[0]);
    tval = (tbuf[1]<<16) + (tbuf[2]<<8) + tbuf[3];
    printf("Number of blocks        = %ld\n",tval);
    tval = (tbuf[5]<<16) + (tbuf[6]<<8) + tbuf[7];
    printf("Block size              = %ld\n",tval);

    tbuf += buffer[3];          /* move to page descriptors */

    while ((tbuf - buffer) < buffer[0])
           {

           switch (tbuf[0] & 0x7f)
                   {
                   case 1:
                          printf("\nError Recovery Parameters\n");
                          printf("=========================\n");
                          printf("Page length              = %d\n",(short)tbuf[1]);
                          printf("DISABLE CORRECTION       = %d\n",(short)tbuf[2]&1);
                          printf("DISABLE XFER ON ERROR    = %d\n",(short)(tbuf[2]>>1)&1);
                          printf("POST ERROR               = %d\n",(short)(tbuf[2]>>2)&1);
                          printf("ENABLE EARLY CORRECTION  = %d\n",(short)(tbuf[2]>>3)&1);
                          printf("READ CONTINUOUS          = %d\n",(short)(tbuf[2]>>4)&1);
                          printf("TRANSFER BLOCK           = %d\n",(short)(tbuf[2]>>5)&1);
                          printf("AUTO READ REALLOCATION   = %d\n",(short)(tbuf[2]>>6)&1);
                          printf("AUTO WRITE REALLOCATION  = %d\n",(short)(tbuf[2]>>7)&1);
                          printf("Retry count              = %d\n",(short)tbuf[3]);
                          printf("Correction span          = %d\n",(short)tbuf[4]);
                          printf("Head offset count        = %d\n",(short)tbuf[5]);
                          printf("Data strobe offset count= %d\n",(short)tbuf[6]);
                          printf("Recovery time limit      = %d\n",(short)tbuf[7]);

                          tbuf += tbuf[1]+2;
                          break;

                   case 2:
                          printf("\nDisconnect/Reconnect Control\n");
                          printf("============================\n");
                          printf("Page length              = %d\n",(short)tbuf[1]);
                          printf("Buffer full ratio        = %d\n",(short)tbuf[2]);
                          printf("Buffer empty ratio       = %d\n",(short)tbuf[3]);
```

```
                    tval = (tbuf[4]<<8)+tbuf[5];
                    printf("Bus inactivity limit     = %d\n",tval);
                    tval = (tbuf[6]<<8)+tbuf[7];
                    printf("Disconnect time limit    = %d\n",tval);
                    tval = (tbuf[8]<<8)+tbuf[9];
                    printf("Connect time limit       = %d\n",tval);
                    tval = (tbuf[10]<<8)+tbuf[11];
                    printf("Maximum burst size       = %d\n",tval);
                    printf("Disable disconnection    = %d\n",(short)tbuf[12]&1);

                    tbuf += tbuf[1]+2;
                    break;

               case 3:
                    printf("\nDevice Format Parameters\n");
                    printf("========================\n");
                    printf("Page length                = %d\n",(short)tbuf[1]);
                    tval = (tbuf[2]<<8)+tbuf[3];
                    printf("Tracks per zone          = %d\n",tval);
                    tval = (tbuf[4]<<8)+tbuf[5];
                    printf("Alternate sectors/zone   = %d\n",tval);
                    tval = (tbuf[6]<<8)+tbuf[7];
                    printf("Alternate tracks/zone    = %d\n",tval);
                    tval = (tbuf[8]<<8)+tbuf[9];
                    printf("Alternate tracks/volume = %d\n",tval);
                    tval = (tbuf[10]<<8)+tbuf[11];
                    printf("Sectors per track          = %d\n",tval);
                    tval = (tbuf[12]<<8)+tbuf[13];
                    printf("Bytes per sector           = %d\n",tval);
                    tval = (tbuf[14]<<8)+tbuf[15];
                    printf("Interleave                 = %d\n",tval);
                    tval = (tbuf[16]<<8)+tbuf[17];
                    printf("Track skew factor          = %d\n",tval);
                    tval = (tbuf[18]<<8)+tbuf[19];
                    printf("Cylinder skew factor     = %d\n",tval);

                    tbuf += tbuf[1]+2;
                    break;

               case 4:
                    printf("\nDrive Geometry Parameters\n");
                    printf("=========================\n");
                    printf("Page length                = %d\n",(short)tbuf[1]);
                    tval = (tbuf[2]<<16)+(tbuf[3]<<8)+tbuf[4];
                    printf("Number of cylinders      = %ld\n",tval);
                    printf("Number of heads            = %d\n",(short)tbuf[5]);
                    tval = (tbuf[6]<<16)+(tbuf[6]<<8)+tbuf[8];
                    printf("Start write precomp        = %ld\n",tval);
                    tval = (tbuf[9]<<16)+(tbuf[10]<<8)+tbuf[11];
                    printf("Start reduced write      = %ld\n",tval);
                    tval = (tbuf[12]<<8)+tbuf[13];
                    printf("Drive step rate            = %d\n",tval);
                    tval = (tbuf[14]<<16)+(tbuf[15]<<8)+tbuf[16];
                    printf("Landing zone cylinder    = %ld\n",tval);

                    tbuf += tbuf[1]+2;
                    break;

               default:
                     printf("\nVendor Unique Page Code %2x\n",(short)tbuf[0]);
                     printf("===========================\n");
                     for (i=0; i<=tbuf[1]+1; i++ )
                          printf("%x ",(short)tbuf[i]);

                     printf("\n");
                     tbuf += tbuf[1]+2;
               }
          }
     }

CloseDevice( &SCSIReq );
FreeMem( buffer, BUFSIZE );
FreeSignal(Port.mp_SigBit);
}
```

```
void ShowSenseData(void)
{
int i;

for (i=0; i<18; i++)
     printf("%x ",(int)Sense[i]);

printf("\n");
}
```

Additional Information on the SCSI Device

Additional programming information on the SCSI device can be found in the include files for the SCSI device and RigidDiskBlock. Both are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

For information on the SCSI commands, see either the ANSI-X3T9 (draft SCSI-2) or ANSI X3.131 (SCSI-1) specification. The NCR SCSI BBS—phone number (316)636-8700 (2400 baud)—has electronic copies of the current SCSI specifications.

SCSI Device Information	
INCLUDES	devices/scsidisk.h
	devices/scsidisk.i
	devices/hardblocks.h
	devices/hardblocks.i

chapter twelve
SERIAL DEVICE

The serial device provides a hardware-independent interface to the Amiga's built-in RS-232C compatible serial port. Serial ports have a wide range of uses, including communication with modems, printers, MIDI devices, and other computers. The same device interface can be used for additional "byte stream oriented devices"—usually more serial ports. The serial device is based on the conventions of Exec device I/O, with extensions for parameter setting and control.

Serial Device Characteristics	
MODES	Exclusive Shared Access
BAUD RATES	110-292,000
HANDSHAKING	Three-Wire Seven-Wire

Serial Device Commands and Functions

Device Command	Operation
CMD_CLEAR	Reset the serial port's read buffer pointers.
CMD_FLUSH	Purge all queued requests for the serial device (does not affect active requests).
CMD_READ	Read a stream of characters from the serial port buffer. The number of characters can be specified or a termination character(s) used.
CMD_RESET	Reset the serial port to its initialized state. All active and queued I/O requests will be aborted and the current buffer will be released.
CMD_START	Restart all paused I/O over the serial port. Also sends an "xON".
CMD_STOP	Pause all active I/O over the serial port. Also sends an "xOFF".
CMD_WRITE	Write out a stream of characters to the serial port. The number of characters can be specified or a NULL-terminated string can be sent.
SDCMD_BREAK	Send a break signal out the serial port. May be done immediately or queued. Duration of the break (in microseconds) can be set by the application.
SDCMD_QUERY	Return the status of the serial port lines and registers, and the number of bytes in the serial port's read buffer.
SDCMD_SETPARAMS	Set the parameters of the serial port. This ranges from baud rate to number of microseconds a break will last.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the serial device. If the command is in progress, it is stopped immediately. If it is queued, it is removed from the queue.
BeginIO()	Initiate a command and return immediately (asynchronous request). This is used to minimize the amount of system overhead.
CheckIO()	Determine the current state of an I/O request.
CloseDevice()	Relinquish use of the serial device. All requests must be complete.
DoIO()	Initiate a command and wait for completion (synchronous request).
OpenDevice()	Obtain use of the serial device.
SendIO()	Initiate a command and return immediately (asynchronous request).
WaitIO()	Wait for the completion of an asynchronous request. When the request is complete the message will be removed from your reply port.

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure of type **IOExtSer**. This structure will be used to communicate commands to the serial device.
CreatePort()	Create a signal message port for reply messages from the serial device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete an extended I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The serial device operates like the other Amiga devices. To use it, you must first open the serial device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The I/O request used by the serial device is called **IOExtSer**.

```
struct IOExtSer
{
    struct   IOStdReq IOSer;
    ULONG    io_CtlChar;     /* control characters */
    ULONG    io_RBufLen;     /* length in bytes of serial read buffer */
    ULONG    io_ExtFlags;    /* additional serial flags */
    ULONG    io_Baud;        /* baud rate */
    ULONG    io_BrkTime;     /* duration of break in microseconds */
    struct   iOTArray  io_TermArray;  /* termination character array */
    UBYTE    io_ReadLen;     /* number of bits per read character */
    UBYTE    io_WriteLen;    /* number of bits per write character */
    UBYTE    io_StopBits;    /* number of stopbits for read */
    UBYTE    io_SerFlags;    /* serial device flags */
    UWORD    io_Status;      /* status of serial port and lines */
};
```

See the include file *devices/serial.h* for the complete structure definition.

OPENING THE SERIAL DEVICE

Three primary steps are required to open the serial device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an extended I/O request structure of type **IOExtSer** using **CreateExtIO()**. **CreateExtIO()** will initialize the I/O request to point to your reply port.
- Open the serial device. Call **OpenDevice()**, passing the I/O request.

```
struct MsgPort  *SerialMP;           /* Define storage for one pointer */
struct IOExtSer *SerialIO;          /* Define storage for one pointer */

if (SerialMP=CreatePort(0,0) )
    if (SerialIO=(struct IOExtSer *)
           CreateExtIO(SerialMP,sizeof(struct IOExtSer)) )
        SerialIO->io_SerFlags=SERF_SHARED;  /* Turn on SHARED mode */
        if (OpenDevice(SERIALNAME,0L,(struct IORequest *)SerialIO,0) )
            printf("%s did not open\n",SERIALNAME);
```

During the open, the serial device pays attention to a subset of the flags in the **io_SerFlags** field. The flag bits, SERF_SHARED and SERF_7WIRE, must be set before open. For consistency, the other flag bits should also be properly set. Full descriptions of all flags will be given later.

The serial device automatically fills in default settings for all parameters—stop bits, parity, baud rate, etc. For the default unit, the settings will come from Preferences. You may need to change certain parameters, such as the baud rate, to match your requirements. Once the serial device is opened, all characters received will be buffered, *even if there is no current request for them.*

READING FROM THE SERIAL DEVICE

You read from the serial device by passing an **IOExtSer** to the device with CMD_READ set in **io_Command**, the number of bytes to be read set in **io_Length** and the address of the read buffer set in **io_Data**.

```
#define READ_BUFFER_SIZE 256
char SerialReadBuffer[READ_BUFFER_SIZE]; /* Reserve SIZE bytes of storage */

SerialIO->IOSer.io_Length   = READ_BUFFER_SIZE;
SerialIO->IOSer.io_Data     = (APTR)&SerialReadBuffer[0];
SerialIO->IOSer.io_Command  = CMD_READ;
DoIO((struct IORequest *)SerialIO);
```

If you use this example, your task will be put to sleep waiting until the serial device reads 256 bytes (or terminates early). Early termination can be caused by error conditions such as a break. The number of characters *actually received* will be recorded in the **io_Actual** field of the **IOExtSer** structure you passed to the serial device.

WRITING TO THE SERIAL DEVICE

You write to the serial device by passing an **IOExtSer** to the device with CMD_WRITE set in **io_Command**, the number of bytes to be written set in **io_Length** and the address of the write buffer set in **io_Data**.

To write a NULL-terminated string, set the length to -1; the device will output from your buffer until it encounters and transmits a value of zero (0x00).

```
SerialIO->IOSer.io_Length   = -1;
SerialIO->IOSer.io_Data     = (APTR)"Life is but a dream. ";
SerialIO->IOSer.io_Command  = CMD_WRITE;
DoIO((struct IORequest *)SerialIO);          /* execute write */
```

The length of the request is -1, meaning we are writing a NULL-terminated string. The number of characters sent can be found in **io_Actual.**

CLOSING THE SERIAL DEVICE

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**. When the last close is performed, the device will deallocate all resources and buffers.

All IORequests must be complete before **CloseDevice()**. Abort any pending requests with **AbortIO()**.

```
if (!(CheckIO(SerialIO)))
    {
    AbortIO((struct IORequest *)SerialIO);  /* Ask device to abort request, if pending */
    }
WaitIO((struct IORequest *)SerialIO);       /* Wait for abort, then clean up */
CloseDevice((struct IORequest *)SerialIO);
```

A Simple Serial Port Example

```
/*
 * Simple_Serial.c
 *
 * This is an example of using the serial device.  First, we will attempt
 * to create a message port with CreateMsgPort().  Next, we will attempt
 * to create the IORequest with CreateExtIO().  Then, we will attempt to
 * open the serial device with OpenDevice().  If successful, we will write
 * a NULL-terminated string to it and reverse our steps.  If we encounter
 * an error at any time, we will gracefully exit.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/serial.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void main(void)
{
struct MsgPort *SerialMP;       /* pointer to our message port */
struct IOExtSer *SerialIO;      /* pointer to our IORequest */

/* Create the message port */
if (SerialMP=CreateMsgPort())
    {
    /* Create the IORequest */
    if (SerialIO = (struct IOExtSer *)
                   CreateExtIO(SerialMP,sizeof(struct IOExtSer)))
        {
        /* Open the serial device */
        if (OpenDevice(SERIALNAME,0,(struct IORequest *)SerialIO,0L))

            /* Inform user that it could not be opened */
            printf("Error: %s did not open\n",SERIALNAME);
        else
            {
            /* device opened, write NULL-terminated string */
            SerialIO->IOSer.io_Length   = -1;
            SerialIO->IOSer.io_Data     = (APTR)"Amiga ";
            SerialIO->IOSer.io_Command  = CMD_WRITE;
            if (DoIO((struct IORequest *)SerialIO))     /* execute write */
                printf("Write failed.  Error - %d\n",SerialIO->IOSer.io_Error);

            /* Close the serial device */
            CloseDevice((struct IORequest *)SerialIO);
            }
        /* Delete the IORequest */
        DeleteExtIO(SerialIO);
        }
    else
        /* Inform user that the IORequest could be created */
        printf("Error: Could create IORequest\n");

    /* Delete the message port */
    DeleteMsgPort(SerialMP);
    }
else
    /* Inform user that the message port could not be created */
    printf("Error: Could not create message port\n");
}
```

DoIO() vs. SendIO(). The above example code contains some simplifications. The **DoIO()** function in the example is not always appropriate for executing the CMD_READ or CMD_WRITE commands. **DoIO()** will not return until the I/O request has finished. With serial handshaking enabled, a write request may *never* finish. Read requests will not finish until characters arrive at the serial port. The following sections will demonstrate a solution using the **SendIO()** and **AbortIO()** functions.

Alternative Modes for Serial Input or Output

As an alternative to **DoIO()** you can use an asynchronous I/O request to transmit the command. Asynchronous requests are initiated with **SendIO()**. Your task can continue to execute while the device processes the command. You can occasionally do a **CheckIO()** to see if the I/O has completed. The write request in this example will be processed while the example continues to run:

```
SerialIO->IOSer.io_Length   = -1;
SerialIO->IOSer.io_Data     = (APTR)"Save the whales! ";
SerialIO->IOSer.io_Command  = CMD_WRITE;
SendIO((struct IORequest *)SerialIO);

printf("CheckIO %lx\n",CheckIO((struct IORequest *)SerialIO));
printf("The device will process the request in the background\n");
printf("CheckIO %lx\n",CheckIO((struct IORequest *)SerialIO));
WaitIO((struct IORequest *)SerialIO);    /* Remove message and cleanup */
```

Most applications will want to wait on multiple signals. A typical application will wait for menu messages from Intuition at the same time as replies from the serial device. The following fragment demonstrates waiting for one of three signals. The **Wait()** will wake up if the read request ever finishes, or if the user presses Ctrl-C or Ctrl-F from the Shell. This fragment may be inserted into the above complete example.

```
/* Precalculate a wait mask for the CTRL-C, CTRL-F and message
 * port signals.  When one or more signals are received,
 * Wait() will return.  Press CTRL-C to exit the example.
 * Press CTRL-F to wake up the example without doing anything.
 * NOTE: A signal may show up without an associated message!
 */

WaitMask = SIGBREAKF_CTRL_C|
            SIGBREAKF_CTRL_F|
             1L << SerialMP->mp_SigBit;

SerialIO->IOSer.io_Command  = CMD_READ;
SerialIO->IOSer.io_Length   = READ_BUFFER_SIZE;
SerialIO->IOSer.io_Data     = (APTR)&SerialReadBuffer[0];
SendIO(SerialIO);

printf("Sleeping until CTRL-C, CTRL-F, or serial input\n");

while (1)
       {
       Temp = Wait(WaitMask);
       printf("Just woke up (YAWN!)\n");

       if (SIGBREAKF_CTRL_C & Temp)
           break;

       if (CheckIO(SerialIO) ) /* If request is complete... */
           {
           WaitIO(SerialIO);   /* clean up and remove reply */
           printf("%ld bytes received\n",SerialIO->IOSer.io_Actual);
           break;
           }
       }
```

```
AbortIO(SerialIO);   /* Ask device to abort request, if pending */
WaitIO(SerialIO);    /* Wait for abort, then clean up */
```

WaitIO() vs. Remove(). The **WaitIO()** function is used above, even if the request is already known to be complete. **WaitIO()** on a completed request simply removes the reply and cleans up. The **Remove()** function is *not acceptable* for clearing the reply port; other messages may arrive while the function is executing.

HIGH SPEED OPERATION

The more characters that are processed in each I/O request, the higher the total throughput of the device. The following technique will minimize device overhead for reads:

- Use the SDCMD_QUERY command to get the number of characters currently in the buffer (see the *devices/serial.h* Autodocs for information on SDCMD_QUERY).
- Use **DoIO()** to read all available characters (or the maximum size of your buffer). In this case, **DoIO()** is guaranteed to return without waiting.
- If zero characters are in the buffer, post an asynchronous request (**SendIO()**) for 1 character. When at least one is ready, the device will return it. Now go back to the first step.
- If the user decides to quit the program, **AbortIO()** any pending requests.

USE OF BeginIO() WITH THE SERIAL DEVICE

Instead of transmitting the read command with either **DoIO()** or **SendIO()**, you might elect to use the low level **BeginIO()** interface to a device.

BeginIO() works much like **SendIO()**, i.e., asynchronously, except it gives you control over the quick I/O bit (IOB_QUICK) in the **io_Flags** field. Quick I/O saves the overhead of a reply message, and perhaps the overhead of a task switch. If a quick I/O request is actually completed quickly, the entire command will execute in the context of the caller. See the "Exec: Device Input/Output" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more detailed information on quick I/O.

The device will determine if a quick I/O request will be handled quickly. Most non-I/O commands will execute quickly; read and write commands may or may not finish quickly.

```
SerialIO.IOSer.io_Flags |= IOF_QUICK;  /* Set QuickIO Flag */

BeginIO((struct IORequest *)SerialIO);
if (SerialIO->IOSer.io_Flags & IOF_QUICK )
    /* If flag is still set, I/O was synchronous and is now finished.
     * The IORequest was NOT appended a reply port.  There is no
     * need to remove or WaitIO() for the message.
     */
    printf("QuickIO\n");
else
     /* The device cleared the QuickIO bit.  QuickIO could not happen
      * for some reason; the device processed the command normally.
      * In this case BeginIO() acted exactly like SendIO().
      */
     printf("Regular I/O\n");
WaitIO(SerialIO);
```

The way you read from the device depends on your need for processing speed. Generally the **BeginIO()** route provides the lowest system overhead when quick I/O is possible. However, if quick I/O does not work, the same reply message overhead still exists.

ENDING A READ OR WRITE USING TERMINATION CHARACTERS

Reads and writes from the serial device may terminate early if an error occurs or if an end-of-file (EOF) is sensed. For example, if a break is detected on the line, any current read request will be returned with the error **SerErr_DetectedBreak**. The count of characters read to that point will be in the **io_Actual** field of the request.

You can specify a set of possible end-of-file characters that the serial device is to look for in the input stream or output using the SDCDMD_SETPARAMS command. These are contained in an **io_TermArray** that you provide. **io_TermArray** is used only when the SERF_EOFMODE flag is selected (see the "Serial Flags" sectionbelow).

If EOF mode is selected, each input data character read into or written from the user's data block is compared against those in **io_TermArray**. If a match is found, the **IOExtSer** is terminated as complete, and the count of characters transferred (including the termination character) is stored in **io_Actual**.

To keep this search overhead as efficient as possible, the serial device requires that the array of characters be in descending order. The array has eight bytes and all must be valid (that is, do not pad with zeros unless zero is a valid EOF character). Fill to the end of the array with the lowest value termination character. When making an arbitrary choice of EOF character(s), you will get the quickest response from the lowest value(s) available.

```
/*
 * Terminate_Serial.c
 *
 * This is an example of using a termination array for reads from the serial
 * device. A termination array is set up for the characters Q, E, etx (CTRL-D)
 * and eot (CTRL-C).  The EOFMODE flag is set in io_SerFlags to indicate that
 * we want to use a termination array by sending the SDCMD_SETPARAMS command to
 * the device.  Then, a CMD_READ command is sent to the device with
 * io_Length set to 25.
 *
 * The read will terminate whenever one of the four characters in the termination
 * array is received or when 25 characters have been received.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/serial.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif
```

```
void main(void)
{
struct MsgPort  *SerialMP;          /* Define storage for one pointer */
struct IOExtSer *SerialIO;         /* Define storage for one pointer */

struct IOTArray Terminators =
{
0x51450403,   /* Q E etx eot */
0x03030303    /* fill to end with lowest value */
};

#define READ_BUFFER_SIZE 25
UBYTE ReadBuff[READ_BUFFER_SIZE];
UWORD ctr;

if (SerialMP=CreatePort(0,0) )
    {
    if (SerialIO=(struct IOExtSer *) CreateExtIO(SerialMP,sizeof(struct IOExtSer)))
        {
        if (OpenDevice(SERIALNAME,0L,(struct IORequest *)SerialIO,0) )
            printf("%s did not open\n",SERIALNAME);
        else
            {
             /* Tell user what we are doing */
             printf("\fLooking for Q, E, EOT or ETX\n");

             /* Set EOF mode flag
              * Set the termination array
              * Send SDCMD_SETPARAMS to the serial device
              */
             SerialIO->io_SerFlags |= SERF_EOFMODE;
             SerialIO->io_TermArray = Terminators;
             SerialIO->IOSer.io_Command  = SDCMD_SETPARAMS;
             if (DoIO((struct IORequest *)SerialIO))
                 printf("Set Params failed ");   /* Inform user of error */
             else
                 {
                 SerialIO->IOSer.io_Length   = READ_BUFFER_SIZE;
                 SerialIO->IOSer.io_Data     = (APTR)&ReadBuff[0];
                 SerialIO->IOSer.io_Command  = CMD_READ;
                 if (DoIO((struct IORequest *)SerialIO))     /* Execute Read */
                     printf("Error: Read failed\n");
                 else
                     {
                      /* Display all characters received */
                      printf("\nThese characters were read:\n\t\t\tASCII\tHEX\n");
                      for (ctr=0;ctr<SerialIO->IOSer.io_Actual;ctr++)
                           printf("\t\t\t  %c\t%x\n",ReadBuff[ctr],ReadBuff[ctr]);
                      printf("\nThe actual number of characters read: %d\n",
                                  SerialIO->IOSer.io_Actual);
                      }
                 }
            CloseDevice((struct IORequest *)SerialIO);
            }

        DeleteExtIO((struct IORequest *)SerialIO);
        }
    else
        printf("Error: Could not create IORequest\n");

    DeletePort(SerialMP);
    }
else
    printf("Error: Could not create message port\n");
}
```

The read will terminate before the **io_Length** number of characters is read if a ‘Q’, ‘E’, ETX, or EOT is detected in the serial input stream.

USING SEPARATE READ AND WRITE TASKS

In some cases there are advantages to creating a separate **IOExtSer** for reading and writing. This allows simultaneous operation of both reading and writing. Some users of the device have separate tasks for read and write operations. The sample code below creates a separate reply port and request for writing to the serial device.

```
struct IOExtSer *SerialWriteIO;
struct MsgPort  *SerialWriteMP;

/*
 * If two tasks will use the same device at the same time, it is preferred
 * use two OpenDevice() calls and SHARED mode.  If exclusive access mode
 * is required, then you will need to copy an existing IORequest.
 *
 * Remember that two separate tasks will require two message ports.
 */

SerialWriteMP = CreatePort(0,0);
SerialWriteIO = (struct IOExtSer *)
                CreateExtIO( SerialWriteMP,sizeof(struct IOExtSer) );

if (SerialWriteMP && SerialWriteIO )
    {

    /* Copy over the entire old IO request, then stuff the
     * new Message port pointer.
     */

    CopyMem( SerialIO, SerialWriteIO, sizeof(struct IOExtSer) );
    SerialWriteIO->IOSer.io_Message.mn_ReplyPort = SerialWriteMP;

    SerialWriteIO->IOSer.io_Command  = CMD_WRITE;
    SerialWriteIO->IOSer.io_Length   = -1;
    SerialWriteIO->IOSer.io_Data     = (APTR)"A poet's food is love and fame";
    DoIO(SerialWriteIO);
    }
```

Where's OpenDevice()? This code assumes that the **OpenDevice()** function has already been called. The initialized read request block is copied onto the new write request block.

Setting Serial Parameters (SDCMD_SETPARAMS)

When the serial device is opened, default values for baud rate and other parameters are automatically filled in from the serial settings in Preferences. The parameters may be changed by using the SDCMD_SETPARAMS command. The flags are defined in the include file *devices/serial.h.*

Serial Device Parameters (IOExtSer)

IOExtSer Field Name	Serial Device Parameter It Controls
io_CtlChar	Control characters to use for xON, xOFF, INQ, ACK respectively. Positioned within an unsigned longword in the sequence from low address to high as listed. INQ and ACK handshaking is not currently supported.
io_RBufLen	Recommended size of the buffer that the serial device should allocate for incoming data. For some hardware the buffer size will not be adjustable. Changing the value may cause the device to allocate a new buffer, which might fail due to lack of memory. In this case the old buffer will continue to be used. For the built-in unit, the minimum size is 64 bytes. Out-of-range numbers will be truncated by the device. When you do an SDCMD_SETPARAMS command, the driver senses the difference between its current value and the value of buffer size you request. All characters that may already be in the old buffer will be discarded. Thus it is wise to make sure that you do not attempt buffer size changes (or any change to the serial device, for that matter) while any I/O is actually taking place.
io_ExtFlags	An unsigned long that contains the flags SEXTF_MSPON and SEXTF_MARK. SEXTF_MSPON enables either mark or space parity. SEXTF_MARK selects mark parity (instead of space parity). Unused bits are reserved.
io_Baud	The real baud rate you request. This is an unsigned long value in the range of 1 to 4,294,967,295. The device will reject your baud request if the hardware is unable to support it. For the built-in driver, any baud rate in the range of 110 to about 1 megabaud is acceptable. The built-in driver may round 110 baud requests to 112 baud. Although baud rates above 19,200 are supported by the hardware, software overhead will limit your ability to "catch" every single character that should be received. Output data rate, however, is not software-dependent.
io_BrkTime	If you issue a break command, this variable specifies how long, in microseconds, the break condition lasts. This value controls the break time for all future break commands until modified by another SDCMD_SETPARAMS.

io_TermArray	A byte-array of eight termination characters, must be in descending order.. If the EOFMODE bit is set in the serial flags, this array specifies eight possible choices of character to use as an end of file mark. See the section above titled "Ending A Read Using Termination Characters" and the SDCMD_SETPARAMS summary page in the Autodocs
io_ReadLen	How many bits per read character; typically a value of 7 or 8. Generally must be the same as **io_WriteLen**.
io_WriteLen	How many bits per write character; typically a value of 7 or 8. Generally must be the same as **io_ReadLen**.
io_StopBits	How many stop bits are to be expected when reading a character and to be produced when writing a character; typically 1 or 2. The built-in driver does not allow values above 1 if **io_WriteLen** is larger than 7.
io_SerFlags	See the "Serial Flags" section below.
io_Status	Contains status information filled in by the SDCMD_QUERY command. Break status is cleared by the execution of SDCMD_QUERY.

You set the serial parameters by passing an **IOExtSer** to the device with SDCMD_SETPARAMS set in **io_Command** and with the flags and parameters set to the values you want.

```
SerialIO->io_SerFlags      &= ~SERF_PARTY_ON;   /* set parity off */
SerialIO->io_SerFlags      |= SERF_XDISABLED;   /* set xON/xOFF disabled */
SerialIO->io_Baud           = 9600;             /* set 9600 baud */
SerialIO->IOSer.io_Command  = SDCMD_SETPARAMS;  /* Set params command */
if (DoIO((struct IORequest *)SerialIO))
    printf("Error setting parameters!\n");
```

The above fragment modifies two bits in **io_SerFlags** and changes the baud rate. If the parameters you request are unacceptable or out of range, the SDCMD_SETPARAMS command will fail. You are responsible for checking the error code and informing the user.

Proper Time for Parameter Changes. A parameter change should not be performed while an I/O request is actually being processed because it might invalidate the request handling already in progress. To avoid this, you should use SDCMD_SETPARAMS only when you have no serial I/O requests pending.

SERIAL FLAGS (Bit Definitions For io_SerFlags)

There are additional serial device parameters which are controlled by flags set in the **io_SerFlags** field of the **IOExtSer** structure. The default state of all of these flags is zero. SERF_SHARED and SERF_7WIRE must always be set before **OpenDevice()**. The flags are defined in the include file *devices/serial.h.*

Serial Flags (io_SerFlags)

Flag Name	Effect on Device Operation
SERF_XDISABLED	Disable the XON/XOFF feature. XON/XOFF *must* be disabled during XModem transfers.
SERF_EOFMODE	Set this bit if you want the serial device to check input characters against **io_TermArray** and to terminate the read immediately if an end-of-file character has been encountered. *Note*: this bit may be set and reset directly in the user's **IOExtSer** without a call to SDCMD_SETPARAMS.
SERF_SHARED	Set this bit if you want to allow other tasks to simultaneously access the serial port. The default is exclusive-access. Any number of tasks may have shared access. Only one task may have exclusive access. If someone already has the port for exclusive access, your **OpenDevice()** call will fail. This flag must be set before **OpenDevice()**.
SERF_RAD_BOOGIE	If set, this bit activates high-speed mode. Certain peripheral devices (MIDI, for example) require high serial throughput. Setting this bit high causes the serial device to skip certain of its internal checking code to speed throughput. Use SERF_RAD_BOOGIE only when you have: • Disabled parity checking • Disabled XON/XOFF handling • Use 8-bit character length • Do not wish a test for a break signal Note that the Amiga is a multitasking system and has immediate processing of software interrupts. If there are other tasks running, it is possible that the serial driver may be unable to keep up with high data transfer rates, even with this bit set.
SERF_QUEUEDBRK	If set, every break command that you transmit will be enqueued. This means that all commands will be executed on a FIFO (first in, first out) basis. If this bit is cleared (the default), a break command takes immediate precedence over any serial output already enqueued. When the break command has finished, the interrupted request will continue (if not aborted by the user).
SERF_7WIRE	If set at **OpenDevice()** time, the serial device will use seven-wire handshaking for RS-232-C communications. Default is three-wire (pins 2, 3, and 7).
SERF_PARTY_ODD	If set, selects odd parity. If clear, selects even parity.
SERF_PARTY_ON	If set, parity usage and checking is enabled. Also see the SERF_MSPON bit described under **io_ExtFlags** above.

Querying The Serial Device

You query the serial device by passing an **IOExtSer** to the device with SDCMD_QUERY set in **io_Command**. The serial device will respond with the status of the serial port lines and registers, and the number of unread characters in the read buffer.

```
UWORD Serial_Status;
ULONG Unread_Chars;

SerialIO->IOSer.io_Command  = SDCMD_QUERY; /* indicate query */
SendIO((struct IORequest *)SerialIO);

Serial_Status = SerialIO->io_Status; /* store returned status */
Unread_Chars = SerialIO->IOSer.io_Actual; /* store unread count */
```

The 16 status bits of the serial device are returned in **io_Status**; the number of unread characters is returned in **io_Actual**.

Serial Device Status Bits

Bit	Active	Symbol	Function
0	—		Reserved
1	—		Reserved
2	high	(RI)	Parallel Select on the A1000. On the A500 and A2000, Select is also connected to the serial port's Ring Indicator. (Be cautious when making cables.)
3	low	(DSR)	Data set ready
4	low	(CTS)	Clear to send
5	low	(CD)	Carrier detect
6	low	(RTS)	Ready to send
7	low	(DTR)	Data terminal ready
8	high		Read overrun
9	high		Break sent
10	high		Break received
11	high		Transmit x-OFFed
12	high		Receive x-OFFed
13-15	—		(reserved)

Sending the Break Command

You send a break through the serial device by passing an **IOExtSer** to the device with SDCMD_BREAK set in **io_Command**. The break may be immediate or queued. The choice is determined by the state of flag SERF_QUEUEDBRK in **io_SerFlags**.

```
SerialIO->IOSer.io_Command  = SDCMD_BREAK; /* send break */
SendIO((struct IORequest *)SerialIO);
```

The duration of the break (in microseconds) can be set in **io_BrkTime**. The default is 250,000 microseconds (.25 seconds).

Error Codes from the Serial Device

The serial device returns error codes whenever an operation is attempted.

```
SerialIO->IOSer.io_Command  = SDCMD_SETPARAMS; /* Set parameters */
if (DoIO((struct IORequest *)SerialIO))
    printf("Set Params failed. Error: %d ",SerialIO->IOSer.io_Error);
```

The error is returned in the **io_Error** field of the **IOExtSer** structure.

Serial Device Error Codes

Error	Value	Explanation
SerErr_DevBusy	1	Device in use
SerErr_BaudMismatch	2	Baud rate not supported by hardware
SerErr_BufErr	4	Failed to allocate new read buffer
SerErr_InvParam	5	Bad parameter
SerErr_LineErr	6	Hardware data overrun
SerErr_ParityErr	9	Parity error
SerErr_TimerErr	11	Timeout (if using 7-wire handshaking)
SerErr_BufOverflow	12	Read buffer overflowed
SerErr_NoDSR	13	No Data Set Ready
SerErr_DetectedBreak	15	Break detected
SerErr_UnitBusy	16	Selected unit already in use

Multiple Serial Port Support

Applications that use the serial port should provide the user with a means to select the name and unit number of the driver. The defaults will be "serial.device" and unit number 0. Typically unit 0 refers to the user-selected default. Unit 1 refers to the built-in serial port. Numbers above 1 are for extended units. The physically lowest connector on a board will always have the lowest unit number.

Careful attention to error handling is required to survive in a multiple port environment. Differing serial hardware will have different capabilities. The device will refuse to open non-existent unit numbers (symbolic name mapping of unit numbers is not provided at the device level). The SDCMD_SETPARAMS command will fail if the underlying hardware cannot support your parameters. Some devices may use quick I/O for read or write requests, others will not. Watch out for partially completed read requests; **io_Actual** may not match your requested read length.

If the Tool Types mechanism is used for selecting the device and unit, the defaults of "DEVICE=serial.device" and "UNIT=0" should be provided. The user should be able to permanently set the device and unit in a configuration file.

Taking Over the Hardware

For some applications use of the device driver interface is not possible. By following the established rules, applications may take over the serial interface at the hardware level. This extreme step is not, however, encouraged. Taking over means losing the ability to work with additional serial ports, and will limit future compatibility.

Access to the hardware registers is controlled by the **misc.resource**. See the "Resources" chapter, and *exec/misc.i* for details. The MR_SERIALBITS and MR_SERIALPORT units control the serial registers.

One additional complication exists. The current serial device will not release the misc.resource bits until after an expunge. This code provides a work around:

```
/*
* A safe way to expunge ONLY a certain device.
* This code attempts to flush ONLY the named device out of memory and
* nothing else.  If it fails, no status is returned (the information
* would have no valid use after the Permit().
*/
#include <exec/types.h>
#include <exec/execbase.h>

void FlushDevice(char *);

extern struct ExecBase *SysBase;

void FlushDevice(name)
char  *name;
{
struct Device *devpoint;

Forbid();   /* ugly */
if (devpoint = (struct Device *)FindName(&SysBase->DeviceList,name) )
    RemDevice(devpoint);
Permit();
}
```

Advanced Example of Serial Device Usage

```
/*
 * Complex_Serial.c
 *
 * Complex tricky example of serial.device usage
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/serial.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif


void main(void)
{
struct MsgPort  *SerialMP;          /* Define storage for one pointer */
struct IOExtSer *SerialIO;         /* Define storage for one pointer */

#define READ_BUFFER_SIZE 32
char SerialReadBuffer[READ_BUFFER_SIZE]; /* Reserve SIZE bytes of storage */

struct IOExtSer *SerialWriteIO = 0;
struct MsgPort  *SerialWriteMP = 0;

ULONG Temp;
ULONG WaitMask;

if (SerialMP=CreatePort(0,0) )
    {
    if (SerialIO=(struct IOExtSer *)
                 CreateExtIO(SerialMP,sizeof(struct IOExtSer)) )
        {
        SerialIO->io_SerFlags=0;    /* Example of setting flags */

        if (OpenDevice(SERIALNAME,0L,SerialIO,0) )
            printf("%s did not open\n",SERIALNAME);
        else
            {
            SerialIO->IOSer.io_Command  = SDCMD_SETPARAMS;
            SerialIO->io_SerFlags      &= ~SERF_PARTY_ON;
            SerialIO->io_SerFlags      |= SERF_XDISABLED;
            SerialIO->io_Baud           = 9600;
            if (Temp=DoIO(SerialIO))
                printf("Error setting parameters - code %ld!\n",Temp);

            SerialIO->IOSer.io_Command  = CMD_WRITE;
            SerialIO->IOSer.io_Length   = -1;
            SerialIO->IOSer.io_Data     = (APTR)"Amiga.";
            SendIO(SerialIO);
            printf("CheckIO %lx\n",CheckIO(SerialIO));
            printf("The device will process the request in the background\n");
            printf("CheckIO %lx\n",CheckIO(SerialIO));
            WaitIO(SerialIO);

            SerialIO->IOSer.io_Command  = CMD_WRITE;
            SerialIO->IOSer.io_Length   = -1;
            SerialIO->IOSer.io_Data     = (APTR)"Save the whales! ";
            DoIO(SerialIO);             /* execute write */
```

```
SerialIO->IOSer.io_Command  = CMD_WRITE;
SerialIO->IOSer.io_Length   = -1;
SerialIO->IOSer.io_Data     = (APTR)"Life is but a dream.";
DoIO(SerialIO);            /* execute write */

SerialIO->IOSer.io_Command  = CMD_WRITE;
SerialIO->IOSer.io_Length   = -1;
SerialIO->IOSer.io_Data     = (APTR)"Row, row, row your boat.";
SerialIO->IOSer.io_Flags = IOF_QUICK;
BeginIO(SerialIO);

if (SerialIO->IOSer.io_Flags & IOF_QUICK )
    {

    /*
     * Quick IO could not happen for some reason; the device processed
     *  the command normally.  In this case BeginIO() acted exactly
     * like SendIO().
     */

    printf("Quick IO\n");
    }
else
    {

    /* If flag is still set, IO was synchronous and is now finished.
     * The IO request was NOT appended a reply port.  There is no
     * need to remove or WaitIO() for the message.
     */

    printf("Regular IO\n");
    }

WaitIO(SerialIO);


SerialIO->IOSer.io_Command  = CMD_UPDATE;
SerialIO->IOSer.io_Length   = -1;
SerialIO->IOSer.io_Data     = (APTR)"Row, row, row your boat.";
SerialIO->IOSer.io_Flags = IOF_QUICK;
BeginIO(SerialIO);

if (0 == SerialIO->IOSer.io_Flags & IOF_QUICK )
    {

    /*
     * Quick IO could not happen for some reason; the device processed
     * the command normally.  In this case BeginIO() acted exactly
     * like SendIO().
     */

    printf("Regular IO\n");

    WaitIO(SerialIO);
    }
else
    {

    /* If flag is still set, IO was synchronous and is now finished.
     * The IO request was NOT appended a reply port.  There is no
     * need to remove or WaitIO() for the message.
     */

    printf("Quick IO\n");
    }


/* Precalculate a wait mask for the CTRL-C, CTRL-F and message
 * port signals.  When one or more signals are received,
 * Wait() will return.  Press CTRL-C to exit the example.
 * Press CTRL-F to wake up the example without doing anything.
 * NOTE: A signal may show up without an associated message!
 */

WaitMask = SIGBREAKF_CTRL_C|
            SIGBREAKF_CTRL_F|
             1L << SerialMP->mp_SigBit;
            SerialIO->IOSer.io_Command  = CMD_READ;
            SerialIO->IOSer.io_Length   = READ_BUFFER_SIZE;
            SerialIO->IOSer.io_Data     = (APTR)&SerialReadBuffer[0];
            SendIO(SerialIO);

            printf("Sleeping until CTRL-C, CTRL-F, or serial input\n");

            while (1)
                   {
                   Temp = Wait(WaitMask);
                   printf("Just woke up (YAWN!)\n");

                   if (SIGBREAKF_CTRL_C & Temp)
                       break;

                   if (CheckIO(SerialIO) ) /* If request is complete... */
                       {
                       WaitIO(SerialIO);   /* clean up and remove reply */

                       printf("%ld bytes received\n",SerialIO->IOSer.io_Actual);
                       break;
                       }
                   }

            AbortIO(SerialIO);  /* Ask device to abort request, if pending */
            WaitIO(SerialIO);   /* Wait for abort, then clean up */


            /*
             * If two tasks will use the same device at the same time, it is preferred
             * use two OpenDevice() calls and SHARED mode.  If exclusive access mode
             * is required, then you will need to copy an existing IO request.
             *
             * Remember that two separate tasks will require two message ports.
             */

            SerialWriteMP = CreatePort(0,0);
            SerialWriteIO = (struct IOExtSer *)
                            CreateExtIO( SerialWriteMP,sizeof(struct IOExtSer) );

            if (SerialWriteMP && SerialWriteIO )
                {

                /* Copy over the entire old IO request, then stuff the
                 * new Message port pointer.
                 */

                CopyMem( SerialIO, SerialWriteIO, sizeof(struct IOExtSer) );
                SerialWriteIO->IOSer.io_Message.mn_ReplyPort = SerialWriteMP;

                SerialWriteIO->IOSer.io_Command  = CMD_WRITE;
                SerialWriteIO->IOSer.io_Length   = -1;
                SerialWriteIO->IOSer.io_Data     = (APTR)"A poet's food is love and fame";
                DoIO(SerialWriteIO);
                }

            if (SerialWriteMP)
                DeletePort(SerialWriteMP);

            if (SerialWriteIO)
                DeleteExtIO(SerialWriteIO);

            CloseDevice(SerialIO);
            }

        DeleteExtIO(SerialIO);
        }
    else
        printf("Unable to create IORequest\n");

    DeletePort(SerialMP);
    }
else
    printf("Unable to create message port\n");
}
```

Additional Information on the Serial Device

Additional programming information on the serial device can be found in the include files and the Autodocs for the serial device. Both are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Serial Device Information	
INCLUDES	devices/serial.h devices/serial.i
AUTODOCS	serial.doc

chapter thirteen
TIMER DEVICE

The Amiga timer device provides a general interface to the Amiga's internal clocks. Through the timer device, time intervals can be measured, time delays can be effected, system time can be set and retrieved, and arithmetic operations can be performed on time values.

New Timer Features for Version 2.0	
Feature	**Description**
UNIT_ECLOCK	New timer device unit
UNIT_WAITUNTIL	New timer device unit
UNIT_WAITECLOCK	New timer device unit
ReadEClock()	New function

Compatibility Warning: The new features for 2.0 are not backwards compatible.

Timer Device Commands and Functions

Command	Operation
TR_ADDREQUEST	Request that the timer device wait a specified period of time before replying to the request.
TR_GETSYSTIME	Get system time and place in a **timeval** structure.
TR_SETSYSTIME	Set the system time from the value in a **timeval** structure.

Device Functions

AddTime()	Add one **timeval** structure to another. The result is placed in the first **timeval** structure.
CmpTime()	Compare one **timeval** structure to another. The result is returned as a longword.
GetSysTime()	Get system time and place in a **timeval** structure.
ReadEClock()	Read the current 64 bit value of the E-Clock into an **EClockVal** structure. The count rate of the E-Clock is also returned. (V36)
SubTime()	Subtract one **timerequest** structure from another. The result is placed in the first **timerequest** structure.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the timer device.
CheckIO()	Return the status of an I/O request.
CloseDevice()	Relinquish use of the timer device. All requests must be complete before closing.
DoIO()	Initiate a command and wait for completion (synchronous request).
OpenDevice()	Obtain use of the timer device. The timer device may be opened multiple times.
SendIO()	Initiate a command and return immediately (asynchronous request).

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure of type **timerequest**. This structure will be used to communicate commands to the timer device.
CreatePort()	Create a signal message port for reply messages from the timer device. Exec will signal a task when a message arrives at the reply port.
DeleteExtIO()	Delete the **timerequest** extended I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The timer device operates in a similar manner to the other Amiga devices. To use it, you must first open it, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The timer device also provides timer functions in addition to the usual I/O request protocol. These functions still require the device to be opened with the proper timer device unit, but do not require a message port. However, the base address of the timer library must be obtained in order to use the timer functions.

The two modes of timer device operation are not mutually exclusive. You may use them both within the same application.

The I/O request used by the timer device is called **timerequest**.

```
struct timerequest
{
    struct IORequest tr_node;
    struct timeval tr_time;
};
```

The timer device functions are passed a time structure, either **timeval** for non E-Clock units or **EClockVal** for E-Clock units.

```
struct timeval
{
     ULONG tv_secs;    /* seconds */
     ULONG tv_micro;   /* microseconds */
};

struct EClockVal
{
    ULONG ev_hi;    /* Upper longword of E-Clock time */
    ULONG ev_lo;    /* Lower longword of E-Clock time */
};
```

See the include file *devices/timer.h* for the complete structure definitions. Time requests fall into three categories:

- Time delay - wait a specified period of time. A time delay causes an application to wait a certain length of time. When a time delay is requested, the number of seconds and microseconds to delay are specified in the I/O request.
- Time measure - how long something takes to complete. A time measure is a three-step procedure where the system or E-Clock time is retrieved, an operation or series of operations is performed, and then another time retrieval is done. The difference between the two time values is the measure of the duration of the operation.
- Time alarm - wait till a specific time. A time alarm is a request to be notified when a specific time value has occurred. It is similar to a time delay except that the absolute time value is specified in the I/O request.

What is an E-Clock? The E-Clock is the clock used by the Motorola 68000 processor family to communicate with other Motorola 8-bit chips. The E-Clock returns two distinct values—the E-Clock value in the form of two longwords and the count rate (tics/second) of the E-Clock. The count rate is related to the master frequency of the machine and is different between PAL and NTSC machines.

TIMER DEVICE UNITS

There are five units in the timer device.

Timer Device Units	
Unit	**Use**
UNIT_MICROHZ	Interval Timing
UNIT_VBLANK	Interval Timing
UNIT_ECLOCK	Interval Timing
UNIT_WAITUNTIL	Time Event Occurrence
UNIT_WAITECLOCK	Time Event Occurrence

- The VBLANK timer unit is very stable and has a granularity comparable to the vertical blanking time. When you make a timing request, such as "signal me in 21 seconds," the reply will come at the next vertical blank after 21 seconds have elapsed. This timer has very low overhead and may be more accurate then the MICROHZ and ECLOCK units for long time periods. *Keep in mind that the vertical blanking time varies depending on the display mode.*
- The MICROHZ timer unit uses the built-in precision hardware timers to create the timing interval you request. It accepts the same type of command—"signal me in so many seconds and microseconds." The microhertz timer has the advantage of greater resolution than the vertical blank timer, but it may have less accuracy over long periods of time. The microhertz timer also has much more system overhead, which means accuracy is reduced as the system load increases. It is primarily useful for short-burst timing for which critical accuracy is not required.
- The ECLOCK timer unit uses the Amiga E-Clock to measure the time interval you request. This is the most precise time measure available through the timer device.
- The WAITUNTIL timer unit acts as an alarm clock for time requests. It will signal the task when systime is greater than or equal to a specified time value. It has the same granularity as the VBLANK timer unit.
- The WAITECLOCK timer unit acts as an alarm clock for time requests. It will signal the task when the E-Clock value is greater than or equal to a specified time value. It has the same granularity as the ECLOCK timer unit.

Granularity vs. Accuracy. Granularity is the sampling frequency used to check the timers. Accuracy is the precision of a measured time interval with respect to the same time interval in real-time. We speak only of granularity because the sampling frequency directly affects how accurate the timers appear to be.

OPENING THE TIMER DEVICE

Three primary steps are required to open the timer device:

- Create a message port using **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an I/O request structure of type **timerequest** using **CreateExtIO()**.
- Open the timer device with one of the five timer device units. Call **OpenDevice()** passing a pointer to the **timerequest**.

```
struct MsgPort *TimerMP;    /* Message port pointer */
struct timerequest *TimerIO;  /* I/O structure pointer */

  /* Create port for timer device communications */
if (!(TimerMP = CreatePort(0,0)))
    cleanexit(" Error: Can't create port\n",RETURN_FAIL);

  /* Create message block for device IO */
if (!(TimerIO = (struct timerequest *)
                CreateExtIO(TimerMP)(sizeof timerequest)) )
    cleanexit(" Error: Can't create IO request\n",RETURN_FAIL);

  /* Open the timer device with UNIT_MICROHZ */
if (error=OpenDevice(TIMERNAME,UNIT_MICROHZ,TimerIO,0))
    cleanexit(" Error: Can't open Timer.device\n",RETURN_FAIL);
```

The procedure for applications which only use the timer device functions is slightly different:

- Declare the timer device base address variable **TimerBase** in the global data area.
- Allocate memory for a **timerequest** structure and a **timeval** structure using **AllocMem()**.
- Call **OpenDevice()**, passing the allocated **timerequest** structure.
- Set the timer device base address variable to point to the timer device base.

```
struct Library *TimerBase;  /* global library pointer */

struct timerequest *TimerIO;
struct timeval *time1;

  /* Allocate memory for timerequest and timeval structures */
TimerIO=(struct timerequest *)AllocMem(sizeof(struct timerequest),
                               MEMF_PUBLIC | MEMF_CLEAR);
time1=(struct timeval *)AllocMem(sizeof(struct timeval),
                               MEMF_PUBLIC | MEMF_CLEAR);
if (!TimerIO | !time1)
    cleanexit(" Error: Can't allocate memory for I/O structures\n",RETURN_FAIL);

if (error=OpenDevice(TIMERNAME,UNIT_MICROHZ,TimerIO,0))
    cleanexit(" Error: Can't open Timer.device\n",RETURN_FAIL);

  /* Set up pointer for timer functions */
TimerBase = (struct Library *)TimerIO->tr_node.io_Device;
```

CLOSING THE TIMER DEVICE

Each **OpenDevice()** must eventually be matched by a call to **CloseDevice()**.

All I/O requests must be complete before **CloseDevice()**. If any requests are still pending, abort them with **AbortIO()**.

```
if (!(CheckIO(TimerIO)))
    {
    AbortIO(TimerIO);       /* Ask device to abort any pending requests */
    }
WaitIO(TimerIO);           /* Clean up */
CloseDevice((struct IORequest *)TimerIO);  /* Close Timer device */
```

System Time

The Amiga has a *system time* feature provided for the convenience of the developer. It is a monotonically increasing time base which should be the same as real time. The timer device provides two commands to use with the system time. In addition, there are utility functions in utility.library which are very useful with system time. See the "Utilities Library" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

The command TR_SETSYSTIME sets the system's idea of what time it is. The system starts out at time "zero" so it is safe to set it forward to the "real" time. However, care should be taken when setting the time backwards.

The command TR_GETSYSTIME is used to get the system time. The timer device does not interpret system time to any physical value. By convention, it tells how many seconds have passed since midnight, January 1, 1978. Your program must calculate the time from this value.

The function **GetSysTime()** can also be used to get the system time. It returns the same value as TR_GETSYSTIME, but uses less overhead.

Whenever someone asks what time it is using TR_GETSYSTIME, the return value of the system time is guaranteed to be unique and unrepeating so that it can be used by applications as a unique identifier.

> *System time at boot time.* The timer device sets system time to zero at boot time. AmigaDOS will then reset the system time to the value specified on the boot disk. If the AmigaDOS *C:SetClock* command is given, this also resets system time.

Here is a program that can be used to determine the system time. The command is executed by the timer device and, on return, the caller can find the data in his request block.

```
/* Get_Systime.c
 *
 * Get system time example
 *
 * Compile with SAS C 5.10: LC -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <devices/timer.h>
```

```
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct timerequest *TimerIO;
struct MsgPort *TimerMP;
struct Message *TimerMSG;

VOID main(VOID);

void main()
{
LONG error;
ULONG days,hrs,secs,mins,mics;

if (TimerMP = CreatePort(0,0))
    {
    if (TimerIO = (struct timerequest *)
                   CreateExtIO(TimerMP,sizeof(struct timerequest)) )
        {
            /* Open with UNIT_VBLANK, but any unit can be used */
        if (!(error=OpenDevice(TIMERNAME,UNIT_VBLANK,(struct IORequest *)TimerIO,0L)))
            {
            /* Issue the command and wait for it to finish, then get the reply */
            TimerIO->tr_node.io_Command = TR_GETSYSTIME;
            DoIO((struct IORequest *) TimerIO);

            /* Get the results and close the timer device */
            mics=TimerIO->tr_time.tv_micro;
            secs=TimerIO->tr_time.tv_secs;

            /* Compute days, hours, etc. */
            mins=secs/60;
            hrs=mins/60;
            days=hrs/24;
            secs=secs%60;
            mins=mins%60;
            hrs=hrs%24;

            /* Display the time */
            printf("\nSystem Time (measured from Jan.1,1978)\n");
            printf("  Days   Hours  Minutes Seconds Microseconds\n");
            printf("%6ld %6ld %6ld %6ld %10ld\n",days,hrs,mins,secs,mics);

            /* Close the timer device */
            CloseDevice((struct IORequest *) TimerIO);
            }
        else
            printf("\nError: Could not open timer device\n");

        /* Delete the IORequest structure */
        DeleteExtIO(TimerIO);
        }
    else
        printf("\nError: Could not create I/O structure\n");

    /* Delete the port */
    DeletePort(TimerMP);
    }
else
    printf("\nError: Could not create port\n");
}
```

Adding a Time Request

Time delays and time alarms are done by opening the timer device with the proper unit and submitting a **timerequest** to the device with TR_ADDREQUEST set in **io_Command** and the appropriate values set in **tv_secs** and **tv_micro**.

Time delays are used with the UNIT_MICROHZ, UNIT_VBLANK, and UNIT_ECLOCK units. The time specified in a time delay **timerequest** is a relative measure from the time the request is posted. This means that the **tv_secs** and **tv_micro** fields should be set to the amount of delay required.

When the specified amount of time has elapsed, the driver will send the **timerequest** back via **ReplyMsg()**. You must fill in the **ReplyPort** pointer of the **timerequest** structure if you wish to be signaled. Also, the number of microseconds must be normalized; it should be a value less than one million.

For a minute and a half delay, set 60 in **tv_secs** and 500,000 in **tv_micro**.

```
TimerIO->tr_node.io_Command = TR_ADDREQUEST;
TimerIO->tr_time.tv_secs  = 60;          /* Delay a minute */
TimerIO->tr_time.tv_micro = 500000;      /* and a half     */
DoIO(TimerIO);
```

Time alarms are used with the UNIT_WAITUNTIL and UNIT_WAITECLOCK units. The **tv_secs** and **tv_micro** fields should be set to the absolute time value of the alarm. For an alarm at 10:30 tonight, the number of seconds from midnight, January 1, 1978 till 10:30 tonight should be set in **tv_secs**. The timer device will not return until the time is greater than or equal to the absolute time value.

For our purposes, we will set an alarm for three hours from now by getting the current system time and adding three hours of seconds to it.

```
#define SECSPERHOUR (60*60)
struct timeval *systime;

GetSysTime(systime);   /* Get current system time */

TimerIO->tr_node.io_Command = TR_ADDREQUEST;
TimerIO->tr_time.tv_secs  = systime.tv_secs+(SECSPERHOUR*3); /* Alarm in 3 hours */
TimerIO->tr_time.tv_micro = systime.tv_micro;
DoIO(TimerIO);
```

> *Time requests with the E-Clock Units.* Time requests with the E-Clock units—UNIT_ECLOCK and UNIT_WAITECLOCK—work the same as the other units except that the values specified in their I/O requests are compared against the value of the E-Clock. See the section "E-Clock Time and Its Relationship to Actual Time" below.

Remember, you must *never* reuse a **timerequest** until the timer device has replied to it. When you submit a timer request, the driver destroys the values you have provided in the **timeval** structure. This means that you must reinitialize the time specification before reposting a **timerequest**.

Keep in mind that the timer device provides a general time-delay capability. It can signal you when *at least* a certain amount of time has passed. The timer device is very accurate under normal system loads, but because the Amiga is a multitasking system, the timer device cannot guarantee that exactly the specified amount of time has elapsed—processing overhead increases as more tasks are run. High-performance applications (such as MIDI time-stamping) may want to take over the 16-bit counters of the CIA B timer resource instead of using the timer device.

> *Problems with small time requests in V1.3 and earlier versions.* You must also take care to avoid posting a **timerequest** of less than 2 microseconds with the UNIT_MICROHZ timer device if you are using V1.3 or earlier versions of the system software. *In V1.3 and earlier versions of the Amiga system software, sending a timerequest for 0 or 1 microseconds can cause a system crash.* Make sure all your timer requests are for 2 microseconds or more when you use the UNIT_MICROHZ timer with those versions.

MULTIPLE TIMER REQUESTS

Multiple requests may be posted to the timer driver. For example, you can make three timer requests in a row:

Signal me in 20 seconds (request 1)
Signal me in 30 seconds (request 2)
Signal me in 10 seconds (request 3)

As the timer queues these requests, it changes the time values and sorts the timer requests to service each request at the desired interval, resulting effectively in the following order:

(request 3) in now+10 seconds
(request 1) 10 seconds after request 3 is satisfied
(request 2) 10 seconds after request 1 is satisfied

If you wish to send out multiple timer requests, you have to create multiple request blocks. You can do this by allocating memory for each **timerequest** you need and filling in the appropriate fields with command data. Some fields are initialized by the call to the **OpenDevice()** function. So, for convenience, you may allocate memory for the **timerequests** you need, call **OpenDevice()** with one of them, and then copy the initialized fields into all the other **timerequests**.

It is also permissible to open the timer device multiple times. In some cases this may be easier than opening it once and using multiple requests. When multiple requests are given, **SendIO()** should be used to transmit each one to the timer.

```
/* Multiple_Timers.c
 *
 * This program is designed to do multiple (3) time requests using one
 * OpenDevice.  It creates a message port - TimerMP, creates an
 * extended I/O structure of type timerequest named TimerIO[0] and
 * then uses that to open the device.  The other two time request
 * structures - TimerIO[1] and TimerIO[2] - are created using AllocMem
 * and then copying TimerIO[0] into them.  The tv_secs field of each
 * structure is set and then three SendIOs are done with the requests.
 * The program then goes into a while loop until all messages are received.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */
```

```
#include <exec/types.h>
#include <exec/memory.h>
#include <devices/timer.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

VOID main(VOID);

void main(void)
{
struct timerequest *TimerIO[3];
struct MsgPort *TimerMP;
struct Message *TimerMSG;

ULONG error,x,seconds[3]={4,1,2}, microseconds[3]={0,0,0};

int allin = 3;
char *position[]={"last","second","first"};

if (TimerMP = CreatePort(0,0))
    {
    if (TimerIO[0] = (struct timerequest *)
                      CreateExtIO(TimerMP,sizeof(struct timerequest)) )
        {
            /* Open the device once */
        if (!(error=OpenDevice( TIMERNAME, UNIT_VBLANK,(struct IORequest *) TimerIO[0], 0L)))
            {
            /* Set command to TR_ADDREQUEST */
            TimerIO[0]->tr_node.io_Command = TR_ADDREQUEST;

            if (TimerIO[1]=(struct timerequest *)
                    AllocMem(sizeof(struct timerequest),MEMF_PUBLIC | MEMF_CLEAR))
                {
                if (TimerIO[2]=(struct timerequest *)
                        AllocMem(sizeof(struct timerequest),MEMF_PUBLIC | MEMF_CLEAR))
                    {
                    /* Copy fields from the request used to open the timer device */
                    *TimerIO[1] = *TimerIO[0];
                    *TimerIO[2] = *TimerIO[0];

                    /* Initialize other fields */
                    for (x=0;x<3;x++)
                        {
                        TimerIO[x]->tr_time.tv_secs   = seconds[x];
                        TimerIO[x]->tr_time.tv_micro  = microseconds[x];
                        printf("\nInitializing TimerIO[%d]",x);
                        }

                    printf("\n\nSending multiple requests\n\n");

                    /* Send multiple requests asynchronously */
                    /* Do not got to sleep yet...            */
                    SendIO((struct IORequest *)TimerIO[0]);
                    SendIO((struct IORequest *)TimerIO[1]);
                    SendIO((struct IORequest *)TimerIO[2]);

                    /* There might be other processing done here */

                    /* Now go to sleep with WaitPort() waiting for the requests */
                    while (allin)
                          {
                          WaitPort(TimerMP);
                          /* Get the reply message */
                          TimerMSG=GetMsg(TimerMP);
                          for (x=0;x<3;x++)
                              if (TimerMSG==(struct Message *)TimerIO[x])
                                  printf("Request %ld finished %s\n",x,position[--allin]);
                          }
                    FreeMem(TimerIO[2],sizeof(struct timerequest));
                    }

                else
                    printf("Error: could not allocate TimerIO[2] memory\n");

                FreeMem(TimerIO[1],sizeof(struct timerequest));
                }

            else
                printf("Error could not allocate TimerIO[1] memory\n");

            CloseDevice((struct IORequest *) TimerIO[0]);
            }

        else
            printf("\nError: Could not OpenDevice\n");

        DeleteExtIO((struct IORequest *) TimerIO[0]);
        }

    else
        printf("Error: could not create IORequest\n");

    DeletePort(TimerMP);
    }

else
    printf("\nError: Could not CreatePort\n");
}
```

If all goes according to plan, `TimerIO[1]` will finish first, `TimerIO[2]` will finish next, and `TimerIO[0]` will finish last.

Using the Time Arithmetic Functions

As indicated above, the time arithmetic functions are accessed in the timer device structure as if they were a routine library. To use them, you create an **IORequest** block and open the timer. In the **IORequest** block is a pointer to the device's base address. This address is needed to access each routine as an offset—for example, **_LVOAddTime**, **_LVOSubTime**, **_LVOCmpTime**—from that base address.

```
/*
 * Timer_Arithmetic.c
 *
 * Example of timer device arithmetic functions
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 *
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <devices/timer.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/timer_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
```

```
#endif

struct Library *TimerBase;  /* setup the interface variable (must be global) */

void main(int argc,char **argv)
{
struct timeval     *time1, *time2, *time3;
struct timerequest *tr;
LONG               error,result;

/*------------------------------------*/
/* Get some memory for our structures */
/*------------------------------------*/
time1=(struct timeval *)AllocMem(sizeof(struct timeval),
                                 MEMF_PUBLIC | MEMF_CLEAR);
time2=(struct timeval *)AllocMem(sizeof(struct timeval),
                                 MEMF_PUBLIC | MEMF_CLEAR);
time3=(struct timeval *)AllocMem(sizeof(struct timeval),
                                 MEMF_PUBLIC | MEMF_CLEAR);
tr=(struct timerequest *)AllocMem(sizeof(struct timerequest),
                                 MEMF_PUBLIC | MEMF_CLEAR);
/* Make sure we got the memory */
if(!time1 | !time2 | !time3 | !tr) goto cleanexit;

/*----------------------------------------------------------------------------*/
/* Set up values to test time arithmetic with.  In a real application these   */
/* values might be filled in via the GET_SYSTIME command of the timer device */
/*----------------------------------------------------------------------------*/
time1->tv_secs = 3;   time1->tv_micro = 0;           /* 3.0 seconds */
time2->tv_secs = 2;   time2->tv_micro = 500000;      /* 2.5 seconds */
time3->tv_secs = 1;   time3->tv_micro = 900000;      /* 1.9 seconds */

printf("Time1 is %ld.%ld\n" , time1->tv_secs,time1->tv_micro);
printf("Time2 is %ld.%ld\n" , time2->tv_secs,time2->tv_micro);
printf("Time3 is %ld.%ld\n\n",time3->tv_secs,time3->tv_micro);

/*--------------------------------*/
/* Open the MICROHZ timer device */
/*--------------------------------*/
error = OpenDevice(TIMERNAME,UNIT_MICROHZ,(struct IORequest *) tr, 0L);
if(error) goto cleanexit;

/* Set up to use the special time arithmetic functions */
TimerBase = (struct Library *)tr->tr_node.io_Device;

/*---------------------------------------------------------------------------*/
/* Now that TimerBase is initialized, it is permissible to call the         */
/* time-comparison or time-arithmetic routines.  Result of this example     */
/* is -1 which means the first parameter has greater time value than second */
/* parameter; +1 means the second parameter is bigger; 0 means equal.       */
/*---------------------------------------------------------------------------*/
result = CmpTime( time1, time2 );
printf("Time1 and Time2 compare = %ld\n",result);

/* Add time2 to time1, result in time1 */
AddTime( time1, time2);
printf("Time1 + Time2 = %ld.%ld\n",time1->tv_secs,time1->tv_micro);

/* Subtract time3 from time2, result in time2 */
SubTime( time2, time3);
printf("Time2 - Time3 = %ld.%ld\n",time2->tv_secs,time2->tv_micro);

/*------------------------------------*/
/* Free system resources that we used */
/*------------------------------------*/
cleanexit:
  if (time1)
      FreeMem(time1,sizeof(struct timeval));
  if (time2)
      FreeMem(time2,sizeof(struct timeval));
  if (time3)
      FreeMem(time3,sizeof(struct timeval));
  if (!error)
      CloseDevice((struct IORequest *) tr);
  if (tr)
      FreeMem(tr,sizeof(struct timerequest));
}
```

WHY USE TIME ARITHMETIC?

As mentioned earlier in this section, because of the multitasking capability of the Amiga, the timer device can provide timings that are at least as long as the specified amount of time. If you need more precision than this, using the system timer along with the time arithmetic routines can at least, in the long run, let you synchronize your software with this precision timer after a selected period of time.

Say, for example, that you select timer intervals so that you get 161 signals within each 3-minute span. Therefore, the **timeval** you would have selected would be 180/161, which comes out to 1 second and 118,012 microseconds per interval. Considering the time it takes to set up a call to set the timer and delays due to task-switching (especially if the system is very busy), it is possible that after 161 timing intervals, you may be somewhat beyond the 3-minute time. Here is a method you can use to keep in sync with system time:

1. Begin.
2. Read system time; save it.
3. Perform your loop however many times in your selected interval.
4. Read system time again, and compare it to the old value you saved. (For this example, it will be more or less than 3 minutes as a total time elapsed.)
5. Calculate a new value for the time interval (**timeval**); that is, one that (if precise) would put you exactly in sync with system time the next time around. **Timeval** will be a lower value if the loops took too long, and a higher value if the loops didn't take long enough.
6. Repeat the cycle.

Over the long run, then, your average number of operations within a specified period of time can become precisely what you have designed.

You Can't Do 1+1 on E-Clock Values. The arithmetic functions are not designed to operate on **EClockVals**.

E-Clock Time and Its Relationship to Actual Time

Unlike **GetSysTime()**, the two values returned by **ReadEClock()**—tics/sec and the **EClockVal** structure—have no direct relationship to actual time. The tics/sec is the E-Clock count rate, a value which is related to the system master clock. The **EClockVal** structure is simply the upper longword and lower longword of the E-Clock 64 bit register.

However, when two **EClockVal** structures are subtracted from each other and divided by the tics/sec (which remains constant), the result does have a relationship to actual time. The value of this calculation is a measure of fractions of a second that passed between the two readings.

```
/* E-Clock Fractions of a second fragment
 *
 * This fragment reads the E-Clock twice and subtracts the two ev_lo values
 *        time2->ev_lo  - time1->ev_lo
 * and divides the result by the E-Clock tics/secs returned by ReadEClock()
 * to get the fractions of a second
 */

struct EClockVal *time1,*time2;
ULONG E_Freq;
LONG error;
struct timerequest *TimerIO;

TimerIO  = (struct timerequest *)AllocMem(sizeof(struct timerequest ),
            MEMF_CLEAR | MEMF_PUBLIC);

time1 = (struct EClockVal *)AllocMem(sizeof(struct EClockVal ),
         MEMF_CLEAR | MEMF_PUBLIC);

time2 = (struct EClockVal *)AllocMem(sizeof(struct EClockVal ),
         MEMF_CLEAR | MEMF_PUBLIC);

if (!(error = OpenDevice(TIMERNAME,UNIT_ECLOCK,(struct IORequest *)TimerIO,0L)) )
    {
    TimerBase = (struct Library *)TimerIO->tr_node.io_Device;
    E_Freq =  ReadEClock((struct EClockVal *) time1);   /* Get initial reading */

       /*  place operation to be measured here */

    E_Freq =  ReadEClock((struct EClockVal *) time2);   /* Get second reading */
    printf("\nThe operation took: %f fractions of a second\n",
            (time2->ev_lo-time1->ev_lo)/(double)E_Freq);

    CloseDevice( (struct IORequest *) TimerIO );
    }
```

The Code Takes Some Liberties. The above fragment only uses the lower longword of the **EClockVal** structures in calculating the fractions of a second that passed. This was done to simplify the fragment. Naturally, you would have to at least check the values of the upper longwords if not use them to get an accurate measure.

Example Timer Program

Here is an example program showing how to use the timer device.

```
/* Simple_Timer.c
 *
 * A simple example of using the timer device.
 *
 * Compile with SAS C 5.10: LC -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <devices/timer.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }      /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }   /* really */
#endif

/* Our timer sub-routines */
void delete_timer  (struct timerequest *);
LONG get_sys_time  (struct timeval *);
LONG set_new_time  (LONG);
void wait_for_timer(struct timerequest *, struct timeval *);
LONG time_delay    ( struct timeval *, LONG );
struct timerequest *create_timer( ULONG );
void show_time     (ULONG);

struct Library *TimerBase;      /* to get at the time comparison functions */

/* manifest constants -- "will never change" */
#define   SECSPERMIN   (60)
#define   SECSPERHOUR  (60*60)
#define   SECSPERDAY   (60*60*24)

void main(int argc,char **argv)
{
LONG seconds;
struct timerequest *tr;      /* IO block for timer commands */
struct timeval oldtimeval;   /* timevals to store times     */
struct timeval mytimeval;
struct timeval currentval;

printf("\nTimer test\n");

/* sleep for two seconds */
currentval.tv_secs = 2;
currentval.tv_micro = 0;
time_delay( &currentval, UNIT_VBLANK );
printf( "After 2 seconds delay\n" );

/* sleep for four seconds */
currentval.tv_secs = 4;
currentval.tv_micro = 0;
time_delay( &currentval, UNIT_VBLANK );
printf( "After 4 seconds delay\n" );

/* sleep for 500,000 micro-seconds = 1/2 second */
currentval.tv_secs = 0;
currentval.tv_micro = 500000;
time_delay( &currentval, UNIT_MICROHZ );
printf( "After 1/2 second delay\n" );

printf( "DOS Date command shows: " );
(void) Execute( "date", 0, 0 );
```

```
/* save what system thinks is the time....we'll advance it temporarily */
get_sys_time( &oldtimeval );
printf("Original system time is:\n");
show_time(oldtimeval.tv_secs );

printf("Setting a new system time\n");

seconds = 1000 * SECSPERDAY + oldtimeval.tv_secs;

set_new_time( seconds );

/* (if user executes the AmigaDOS DATE command now, he will*/
/* see that the time has advanced something over 1000 days */
printf( "DOS Date command now shows: " );
(void) Execute( "date", 0, 0 );

get_sys_time( &mytimeval );
printf( "Current system time is:\n");
show_time(mytimeval.tv_secs);

/* Added the microseconds part to show that time keeps */
/* increasing even though you ask many times in a row  */
printf("Now do three TR_GETSYSTIMEs in a row (notice how the microseconds increase)\n\n");
get_sys_time( &mytimeval );
printf("First TR_GETSYSTIME \t%ld.%ld\n",mytimeval.tv_secs, mytimeval.tv_micro);
get_sys_time( &mytimeval );
printf("Second TR_GETSYSTIME \t%ld.%ld\n",mytimeval.tv_secs, mytimeval.tv_micro);
get_sys_time( &mytimeval );
printf("Third TR_GETSYSTIME \t%ld.%ld\n",mytimeval.tv_secs, mytimeval.tv_micro);

printf( "\nResetting to former time\n" );
set_new_time( oldtimeval.tv_secs );

get_sys_time( &mytimeval );
printf( "Current system time is:\n");
show_time(mytimeval.tv_secs);

/* just shows how to set up for using the timer functions, does not */
/* demonstrate the functions themselves.  (TimerBase must have a    */
/* legal value before AddTime, SubTime or CmpTime are performed.    */
tr = create_timer( UNIT_MICROHZ );
TimerBase = (struct Library *)tr->tr_node.io_Device;

/* and how to clean up afterwards */
TimerBase = (struct Library *)(-1);
delete_timer( tr );
}


struct timerequest *create_timer( ULONG unit )
{
/* return a pointer to a timer request.  If any problem, return NULL */
LONG error;
struct MsgPort *timerport;
struct timerequest *TimerIO;

timerport = CreatePort( 0, 0 );
if (timerport == NULL )
    return( NULL );

TimerIO = (struct timerequest *)
    CreateExtIO( timerport, sizeof( struct timerequest ) );
if (TimerIO == NULL )
    {
    DeletePort(timerport);   /* Delete message port */
    return( NULL );
    }

error = OpenDevice( TIMERNAME, unit,(struct IORequest *) TimerIO, 0L );
if (error != 0 )
    {
    delete_timer( TimerIO );
    return( NULL );
    }
return( TimerIO );
}
```

```
/* more precise timer than AmigaDOS Delay() */
LONG time_delay( struct timeval *tv, LONG unit )
{
struct timerequest *tr;
/* get a pointer to an initialized timer request block */
tr = create_timer( unit );

/* any nonzero return says timedelay routine didn't work. */
if (tr == NULL )
    return( -1L );

wait_for_timer( tr, tv );

/* deallocate temporary structures */
delete_timer( tr );
return( 0L );
}


void wait_for_timer(struct timerequest *tr, struct timeval *tv )
{

tr->tr_node.io_Command = TR_ADDREQUEST; /* add a new timer request */

/* structure assignment */
tr->tr_time = *tv;

/* post request to the timer -- will go to sleep till done */
DoIO((struct IORequest *) tr );
}


LONG set_new_time(LONG secs)
{
struct timerequest *tr;
tr = create_timer( UNIT_MICROHZ );

/* non zero return says error */
if (tr == 0 )
    return( -1 );

tr->tr_time.tv_secs = secs;
tr->tr_time.tv_micro = 0;
tr->tr_node.io_Command = TR_SETSYSTIME;
DoIO((struct IORequest *) tr );

delete_timer(tr);
return(0);
}


LONG get_sys_time(struct timeval *tv)
{
struct timerequest *tr;
tr = create_timer( UNIT_MICROHZ );

/* non zero return says error */
if (tr == 0 )
    return( -1 );

tr->tr_node.io_Command = TR_GETSYSTIME;
DoIO((struct IORequest *) tr );

/* structure assignment */
*tv = tr->tr_time;

delete_timer( tr );
return( 0 );
}
```

```
void delete_timer(struct timerequest *tr )
{
struct MsgPort *tp;

if (tr != 0 )
    {
    tp = tr->tr_node.io_Message.mn_ReplyPort;

    if (tp != 0)
        DeletePort(tp);

    CloseDevice( (struct IORequest *) tr );
    DeleteExtIO( (struct IORequest *) tr );
    }
}


void show_time(ULONG secs)
{
ULONG days,hrs,mins;

/* Compute days, hours, etc. */
mins=secs/60;
hrs=mins/60;
days=hrs/24;
secs=secs%60;
mins=mins%60;
hrs=hrs%24;

/* Display the time */
printf("*   Hour Minute Second  (Days since Jan.1,1978)\n");
printf("*%5ld:%5ld:%5ld      (%6ld )\n\n",hrs,mins,secs,days);
}       /* end of main */
```

Additional Information on the Timer Device

Additional programming information on the timer device and the utilities library can be found in their include files and Autodocs. All are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Timer Device Information	
INCLUDES	devices/timer.h
	devices/timer.i
	utility/date.h
	utility/date.i
AUTODOCS	timer.doc
	utility.doc

chapter fourteen
TRACKDISK DEVICE

The Amiga trackdisk device directly drives the disk, controls the disk motors, reads raw data from the tracks, and writes raw data to the tracks. Normally, you use the AmigaDOS functions to write or read data from the disk. The trackdisk device is the lowest-level software access to the disk data and is used by AmigaDOS to access the disks. The trackdisk device supports the usual commands such as CMD_WRITE and CMD_READ. In addition, it supports an extended form of these commands to allow additional control over the trackdisk device.

New Features for Version 2.0	
Feature	**Description**
TD_GETGEOMETRY	Device Command
TD_EJECT	Device Command
IOTF_INDEXSYNC	Device Command Flag
IOTF_WORDSYNC	Device Command Flag
Fast RAM Buffers	Now Supported

Compatibility Warning: The new features for 2.0 are not backwards compatible.

Trackdisk Device Commands and Functions

Command	Operation
CMD_CLEAR ETD_CLEAR	Mark track buffer as invalid. Forces the track to be re-read. ETD_CLEAR also checks for a diskchange.
CMD_READ ETD_READ	Read one or more sectors from a disk. ETD_READ also reads the sector label area and checks for a diskchange.
CMD_UPDATE ETD_UPDATE	Write out track buffer if it has been changed. ETD_UPDATE also checks for a diskchange.
CMD_WRITE ETD_WRITE	Write one or more sectors to a disk. ETD_WRITE also writes the sector label area and checks for a diskchange.
TD_ADDCHANGEINT	Add an interrupt handler to be activated on a diskchange.
TD_CHANGENUM	Return the current value of the diskchange counter used by the ETD commands to determine if a diskchange has occurred.
TD_CHANGESTATE	Return the disk present/not-present status of a drive.
TD_EJECT	Eject a disk from a drive. This command will only work on drives that support an eject command (V36).
TD_FORMAT ETD_FORMAT	Initialize one or more tracks with a data buffer. ETD_FORMAT also initializes the sector label area.
TD_GETDRIVETYPE	Return the type of disk drive in use by the unit.
TD_GETGEOMETRY	Return the disk geometry table (V36).
TD_GETNUMTRACKS	Return the number of tracks usable with the unit.
TD_MOTOR ETD_MOTOR	Turn the motor on or off. ETD_MOTOR also checks for a diskchange.
TD_PROTSTATUS	Return the write-protect status of a disk.
TD_RAWREAD ETD_RAWREAD	Read RAW sector data from disk (unencoded MFM). ETD_RAWREAD also checks for a diskchange.
TD_RAWWRITE ETD_RAWWRITE	Write RAW sector data to disk. ETD_RAWWRITE also checks for a diskchange.
TD_REMCHANGEINT	Remove a diskchange interrupt handler.
TD_SEEK ETD_SEEK	Move the head to a specific track. ETD_SEEK also checks for a diskchange.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the trackdisk device.
BeginIO()	Initiate a command and return immediately (asynchronous request).
CloseDevice()	Relinquish use of a disk unit.
DoIO()	Initiate a command and wait for completion (synchronous request).
OpenDevice()	Obtain exclusive use of a particular disk unit.

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an extended I/O request structure of type **IOExtTD**. This structure will be used to communicate commands to the trackdisk device.
CreatePort()	Create a signal message port for reply messages from the trackdisk device. Exec will signal a task when a message arrives at the reply port.
DeleteExtIO()	Delete an I/O request structure created by **CreateExtIO()**.
DeletePort()	Delete the message port created by **CreatePort()**.

Device Interface

The trackdisk device operates like other Amiga devices. To use it, you must first open the device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The trackdisk device uses two different types of I/O request blocks, **IOStdReq** and **IOExtTD** and two types of commands, standard and extended. An **IOExtTD** is required for the extended trackdisk commands (those beginning with "ETD_"), but can be used for both types of commands. Thus, the **IOExtTD** is the type of I/O request that will be used in this chapter.

```
struct IOExtTD
{
    struct  IOStdReq iotd_Req;
    ULONG   iotd_Count;         /* Diskchange counter */
    ULONG   iotd_SecLabel;      /* Sector label data */
};
```

See the include file *devices/trackdisk.h* for the complete structure definition.

The enhanced commands listed above—those beginning with "ETD_"— are similar to their standard counterparts but have additional features: they allow you to control whether a command will be executed if the disk has been changed and they allow you to read or write to the sector label portion of a sector.

Enhanced commands require a larger I/O request, **IOExtTD**, than the **IOStdReq** request used by the standard commands. **IOExtTD** contains extra information needed by the enhanced command; since the standard form of a command ignores the extra fields, **IOExtTD** requests can be used for both types. The extra information takes the form of two extra longwords at the end of the data structure. These commands are performed only if the change count is less than or equal to the value in the **iotd_Count** field of the command's request block.

The **iotd_Count** field keeps old I/O requests from being performed when the disk is changed. Any request found with an **iotd_Count** less than the current change counter value will be returned with a characteristic error (TDERR_DiskChange) in the **io_Error** field. This allows stale I/O requests to be returned to the user after a disk has been changed. The current disk-change counter value can be obtained by TD_CHANGENUM. If the user wants enhanced disk I/O but does not care about disk removal, then **iotd_Count** may be set to the maximum unsigned long integer value (0xFFFFFFFF).

The **iotd_SecLabel** field allows access to the sector identification section of the sector header. Each sector has 16 bytes of descriptive data space available to it; the trackdisk device does not interpret this data. If **iotd_SecLabel** is NULL, then this descriptive data is ignored. If it is not NULL, then **iotd_SecLabel** should point to a series of contiguous 16-byte chunks (one for each sector that is to be read or written). These chunks will be written out to the sector's label region on a write or filled with the sector's label area on a read. If a CMD_WRITE (the standard write call) is done, then the sector label area is left unchanged.

ABOUT AMIGA FLOPPY DISKS

The standard 3.5 inch Amiga floppy disk consists of a number of tracks that are NUMSECS (11) sectors of TD_SECTOR (512) usable data bytes plus TD_LABELSIZE (16) bytes of label area. There are usually 2 tracks per cylinder (2 heads) and 80 cylinders per disk. The number of tracks can be found using the TD_GETNUMTRACKS command.

For V36 and higher systems, the NUMSECS in some drives may be variable and may change when a disk is inserted. Use TD_GETGEOMETRY to determine the current number of sectors.

> *Think Tracks not Cylinders.* The result is given in tracks and not cylinders. On a standard 3.5" drive, this gives useful space of 880K bytes plus 28K bytes of sector label area per floppy disk.

Although the disk is logically divided up into sectors, all I/O to the disk is done a track at a time. This allows access to the drive with no interleaving and increases the useful storage capacity by about 20 percent. Each disk drive on the system has its own buffer which holds the track data going to and from the drive.

Normally, a read of a sector will only have to copy the data from the track buffer. If the track buffer contains another track's data, then the buffer will first be written back to the disk (if it is "dirty") and the new track will be read in. All track boundaries are transparent to the programmer (except for FORMAT, SEEK, and RAWREAD/RAWWRITE commands) because you give the device an offset into the disk in the number of bytes from the start of the disk. The device ensures that the correct track is brought into memory.

The performance of the disk is greatly enhanced if you make effective use of the track buffer. The performance of sequential reads will be up to an order of magnitude greater than reads scattered across the disk. In addition, only full-sector writes on sector boundaries are supported.

The trackdisk device is based upon a standard device structure. It has the following restrictions:

- All reads and writes must use an **io_Length** that is an integer multiple of TD_SECTOR bytes (the sector size in bytes).
- The offset field must be an integer multiple of TD_SECTOR.
- The data buffer must be word-aligned.
- Under pre-V36, the data buffer must be also be in Chip RAM.

OPENING THE TRACKDISK DEVICE

Three primary steps are required to open the trackdisk device:

- Create a message port by calling **CreatePort()**. Reply messages from the device must be directed to a message port.
- Create an extended I/O request structure of type **IOExtTD**. The **IOExtTD** structure is created by the **CreateExtIO()** function.
- Open the trackdisk device. Call **OpenDevice()**, passing it the extended I/O request.

For the trackdisk device, the flags parameter of the **OpenDevice()** function specifies whether you are opening a 3.5" drive (flags=0) or a 5.25" drive (flags=1). With flags set to 0 trackdisk will only open a 3.5" drive. To tell the device to open any drive it understands, set the flags parameter to TDF_ALLOW_NON_3_5. (See the include file *devices/trackdisk.h* for more information.)

```
#include <devices/trackdisk.h>

struct MsgPort *TrackMP;         /* Pointer for message port */
struct IOExtTD *TrackIO;         /* Pointer for IORequest */

if (TrackMP=CreatePort(0,0) )
    if (TrackIO=(struct IOExtTD *)
           CreateExtIO(TrackMP,sizeof(struct IOExtTD)) )
        if (OpenDevice(TD_NAME,0L,(struct IORequest *)TrackIO,Flags) )
            printf("%s did not open\n",TD_NAME);
```

> *Disk Drive Unit Numbers.* The unit number—second parameter of the **OpenDevice()** call—can be any value from 0 to 3. Unit 0 is the built-in 3.5" disk drive. Units 1 through 3 represent additional disk drives that may be connected to an Amiga system.

READING FROM THE TRACKDISK DEVICE

You read from the trackdisk device by passing an **IOExtTD** to the device with CMD_READ set in **io_Command**, the number of bytes to be read set in **io_Length**, the address of the read buffer set in **io_Data** and the track you want to read—specified as a byte offset from the start of the disk—set in **io_Offset**.

The byte offset of a particular track is calculated by multiplying the number of the track you want to read by the number of bytes in a track. The number of bytes in a track is obtained by multiplying the number of sectors (NUMSECS) by the number of bytes per sector (TD_SECTOR). Thus you would multiply 11 by 512 to get 5632 bytes per track. To read track 15, you would multiply 15 by 5632 giving 84,480 bytes offset from the beginning of the disk.

```
#define TRACK_SIZE ((LONG) (NUMSECS * TD_SECTOR))
UBYTE *Readbuffer;
SHORT tracknum;

if (Readbuffer = AllocMem(TRACK_SIZE,MEMF_CLEAR|MEMF_CHIP))
    {
    DiskIO->iotd_Req.io_Length = TRACK_SIZE;
    DiskIO->iotd_Req.io_Data = (APTR)Readbuffer;
    DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track);
    DiskIO->iotd_Req.io_Command = CMD_READ;
    DoIO((struct IORequest *)DiskIO);
    }
```

For reads using the enhanced read command ETD_READ, the **IOExtTD** is set the same as above with the addition of setting **iotd_Count** to the current diskchange number. The diskchange number is returned by the TD_CHANGENUM command (see below). If you wish to also read the sector label area, you must set **iotd_SecLabel** to a non-NULL value.

```
DiskIO->iotd_Req.io_Length = TRACK_SIZE;
DiskIO->iotd_Req.io_Data = (APTR)Readbuffer;
DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track);
DiskIO->iotd_Count = change_count;
DiskIO->iotd_Req.io_Command = ETD_READ;
DoIO((struct IORequest *)DiskIO);
```

ETD_READ and CMD_READ obey all of the trackdisk device restrictions noted above. They transfer data from the track buffer to the user's buffer. If the desired sector is already in the track buffer, no disk activity is initiated. If the desired sector is not in the buffer, the track containing that sector is automatically read in. If the data in the current track buffer has been modified, it is written out to the disk before a new track is read.

WRITING TO THE TRACKDISK DEVICE

You write to the trackdisk device by passing an **IOExtTD** to the device with CMD_WRITE set in **io_Command**, the number of bytes to be written set in **io_Length**, the address of the write buffer set in **io_Data** and the track you want to write—specified as a byte offset from the start of the disk—set in **io_Offset**.

```
#define TRACK_SIZE ((LONG) (NUMSECS * TD_SECTOR))
UBYTE *Writebuffer;

if (Writebuffer = AllocMem(TRACK_SIZE,MEMF_CLEAR|MEMF_PUBLIC))
    {
    DiskIO->iotd_Req.io_Length = TRACK_SIZE;
    DiskIO->iotd_Req.io_Data = (APTR)Writebuffer;
    DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * tracknum);
    DiskIO->iotd_Req.io_Command = CMD_WRITE;
    DoIO((struct IORequest *)DiskIO);
    }
```

For writes using the enhanced write command ETD_WRITE, the **IOExtTD** is set the same as above with the addition of setting **iotd_Count** to the current diskchange number. The diskchange number is returned by the TD_CHANGENUM command (see below). If you wish to also write the sector label area, you must set **iotd_SecLabel** to a non-NULL value.

```
DiskIO->iotd_Req.io_Length = TRACK_SIZE;
DiskIO->iotd_Req.io_Data = (APTR)Writebuffer;
DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * tracknum);
DiskIO->iotd_Count = change_count;
DiskIO->iotd_Req.io_Command = ETD_WRITE;
DoIO((struct IORequest *)DiskIO);
```

ETD_WRITE and CMD_WRITE obey all of the trackdisk device restrictions noted above. They transfer data from the user's buffer to the track buffer. If the track that contains this sector is already in the track buffer, no disk activity is initiated. If the desired sector is not in the buffer, the track containing that sector is automatically read in. If the data in the current track buffer has been modified, it is written out to the disk before a new track is read in for modification.

CLOSING THE TRACKDISK DEVICE

As with all devices, you *must* close the trackdisk device when you have finished using it. To release the device, a **CloseDevice()** call is executed with the same **IOExtTD** used when the device was opened. This only closes the device and makes it available to the rest of the system. It does **not** deallocate the **IOExtTD** structure.

```
CloseDevice((struct IORequest *)DiskIO);
```

Advanced Commands

DETERMINING THE DRIVE GEOMETRY TABLE

The layout geometry of a disk drive can be determined by using the TD_GETGEOMETRY command. The layout can be defined three ways:

- TotalSectors
- Cylinders and CylSectors
- Cylinders, Heads, and TrackSectors.

Of the three, TotalSectors is the most accurate, Cylinders and CylSectors is less so, and Cylinders, Heads and TrackSectors is the least accurate. All are usable, though the last two may waste some portion of the available space on some drives.

The TD_GETGEOMETRY commands returns the disk layout geometry in a **DriveGeometry** structure:

```
struct DriveGeometry
{
    ULONG dg_SectorSize;        /* in bytes */
    ULONG dg_TotalSectors;      /* total # of sectors on drive */
    ULONG dg_Cylinders;         /* number of cylinders */
    ULONG dg_CylSectors;        /* number of sectors/cylinder */
    ULONG dg_Heads;             /* number of surfaces */
    ULONG dg_TrackSectors;      /* number of sectors/track */
    ULONG dg_BufMemType;        /* preferred buffer memory type */
                                /* (usually MEMF_PUBLIC) */
    UBYTE dg_DeviceType;        /* codes as defined in the SCSI-2 spec*/
    UBYTE dg_Flags;             /* flags, including removable */
    UWORD dg_Reserved;
};
```

See the include file *devices/trackdisk.h* for the complete structure definition and values for the **dg_DeviceType** and **dg_Flags** fields.

You determine the drive layout geometry by passing an **IOExtTD** with TD_GETGEOMETRY set in **io_Command** and a pointer to a **DriveGeometry** structure set in **io_Data**.

```
struct DriveGeometry *Euclid;

Euclid = (struct DriveGeometry *)
          AllocMem(sizeof(struct DriveGeometry),MEMF_PUBLIC | MEMF_CLEAR);

DiskIO->iotd_Req.io_Data = Euclid;     /* put layout geometry here */
DiskIO->iotd_Req.io_Command = TD_GETGEOMETRY;
DoIO((struct IORequest *)DiskIO);
```

For V36 and higher versions of the operating system, TD_GETGEOMETRY is preferred over TD_GETNUMTRACKS for determining the number of tracks on a disk. This is because new drive types may have more sectors or different sector sizes, etc., than standard Amiga drives.

CLEARING THE TRACK BUFFER

ETD_CLEAR and CMD_CLEAR mark the track buffer as invalid, forcing a reread of the disk on the next operation. ETD_UPDATE or CMD_UPDATE would be used to force data out to the disk before turning the motor off. ETD_CLEAR or CMD_CLEAR is usually used after having locked out the trackdisk device via the use of the disk resource, when you wish to prevent the track from being updated, or when you wish to force the track to be re-read. ETD_CLEAR or CMD_CLEAR will not do an update, nor will an update command do a clear.

You clear the track buffer by passing an **IOExtTD** to the device with CMD_CLEAR or ETD_CLEAR set in **io_Command**. For ETD_CLEAR, you must also set **iotd_Count** to the current diskchange number.

```
DiskIO->iotd_Req.io_Command = TD_CLEAR;
DoIO((struct IORequest *)DiskIO);
```

CONTROLLING THE DRIVE MOTOR

ETD_MOTOR and TD_MOTOR give you control of the motor. When the trackdisk device executes this command, the old state of the motor is returned in **io_Actual**. If **io_Actual** is zero, then the motor was off. Any other value implies that the motor was on. If the motor is just being turned on, the device will delay the proper amount of time to allow the drive to come up to speed. Normally, turning the drive on is not necessary—the device does this automatically if it receives a request when the motor is off.

However, turning the motor off *is* the programmer's responsibility. In addition, the standard instructions to the user are that it is safe to remove a disk if, and only if, the motor is off (that is, if the disk light is off).

You control the drive motor by passing an **IOExtTD** to the device with CMD_MOTOR or ETD_MOTOR set in **io_Command** and the state you want to put the motor in set in **io_Length**. If **io_Length** is set to 1, the trackdisk device will turn on the motor; a 0 will turn it off. For ETD_MOTOR, you must also set **iotd_Count** to the current diskchange number.

```
DiskIO->iotd_Req.io_Length = 1;          /* Turn on the drive motor */
DiskIO->iotd_Req.io_Command = TD_MOTOR;
DoIO((struct IORequest *)DiskIO);
```

UPDATING A TRACK SECTOR

The Amiga trackdisk device does not write data sectors unless it is necessary (you request that a different track be used) or until the user requests that an update be performed. This improves system speed by caching disk operations. The update commands ensure that any buffered data is flushed out to the disk. If the track buffer has not been changed since the track was read in, the update commands do nothing.

You update a data sector by passing an **IOExtTD** to the device with CMD_UPDATE or ETD_UPDATE set in **io_Command**. For ETD_UPDATE, you must also set **iotd_Count** to the current diskchange number.

```
DiskIO->iotd_Req.io_Command = TD_UPDATE;
DoIO((struct IORequest *)DiskIO);
```

FORMATTING A TRACK

ETD_FORMAT and TD_FORMAT are used to write data to a track that either has not yet been formatted or has had a hard error on a standard write command. TD_FORMAT completely ignores all data currently on a track and does not check for disk change before performing the command. The device will format the requested tracks, filling each sector with the contents of the buffer pointed to by **io_Data** field. You should do a read pass to verify the data.

If you have a hard write error during a normal write, you may find it possible to use the TD_FORMAT command to reformat the track as part of your error recovery process. ETD_FORMAT will write the sector label area if the **iotd_SecLabel** is non-NULL.

You format a track by passing an **IOExtTD** to the device with CMD_FORMAT or ETD_FORMAT set in **io_Command, io_Data** set to at least track worth of data, **io_Offset** field set to the byte offset of the track you want to write and the **io_Length** set to the length of a track. For ETD_FORMAT, you must also set **iotd_Count** to the current diskchange number.

```
#define TRACK_SIZE ((LONG) (NUMSECS * TD_SECTOR))
UBYTE *Writebuffer;

if (WriteBuffer = AllocMem(TRACK_SIZE,MEMF_CLEAR|MEMF_CHIP))
    {
    DiskIO->iotd_Req.io_Length=TRACK_SIZE;
    DiskIO->iotd_Req.io_Data=(APTR)Writebuffer;
    DiskIO->iotd_Req.io_Offset=(ULONG)(TRACK_SIZE * track);
    DiskIO->iotd_Req.io_Command = TD_FORMAT;
    DoIO((struct IORequest *)DiskIO);
    }
```

EJECTING A DISK

Certain disk drive manufacturers allow software control of disk ejection. The trackdisk device provides the TD_EJECT command to tell such drives to eject a disk.

You eject a disk by passing an **IOExtTD** to the device with TD_EJECT set in **io_Command**.

```
DiskIO->iotd_Req.io_Command = TD_EJECT;
DoIO((struct IORequest *)DiskIO);
```

Read the Instruction Manual. The Commodore 3.5" drives for the Amiga and most other Amiga drive manufacturers do not support software disk ejects. Attempting this command on those drives will result in an error condition. Consult the instruction manual for your disk drive to determine whether this is supported.

Disk Status Commands

Disk status commands return status on the current disk in the opened unit. These commands may be done with quick I/O and thus may be called within interrupt handlers (such as the trackdisk disk change handler). See the "Exec: Device Input/Output" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more detailed information on quick I/O.

DETERMINING THE PRESENCE OF A DISK

You determine the presence of a disk in a drive by passing an **IOExtTD** to the device with TD_CHANGESTATE set in **io_Command**. For quick I/O, you must set **io_Flags** to IOF_QUICK.

```
DiskIO->iotd_Req.io_Flags = IOF_QUICK;
DiskIO->iotd_Req.io_Command = TD_CHANGESTATE;
BeginIO((struct IORequest *)DiskIO);
```

TD_CHANGESTATE returns the presence indicator of a disk in **io_Actual**. The value returned will be zero if a disk is currently in the drive and nonzero if the drive has no disk.

DETERMINING THE WRITE-PROTECT STATUS OF A DISK

You determine the write-protect status of a disk by passing an **IOExtTD** to the device with TD_PROTSTATUS set in **io_Command**. For quick I/O, you must set **io_Flags** to IOF_QUICK.

```
DiskIO->iotd_Req.io_Flags = IOF_QUICK;
DiskIO->iotd_Req.io_Command = TD_PROTSTATUS;
BeginIO((struct IORequest *)DiskIO);
```

TD_PROTSTATUS returns the write-protect status in **io_Actual**. The value will be zero if the disk is not write-protected and nonzero if the disk is write-protected.

DETERMINING THE DRIVE TYPE

You determine the drive type of a unit by passing an **IOExtTD** to the device with TD_GETDRIVETYPE set in **io_Command**. For quick I/O, you must set **io_Flags** to IOF_QUICK.

```
DiskIO->iotd_Req.io_Flags = IOF_QUICK;
DiskIO->iotd_Req.io_Command = TD_GETDRIVETYPE;
BeginIO((struct IORequest *)DiskIO);
```

TD_GETDRIVETYPE returns the drive type for the unit that was opened in **io_Actual**. The value will be **DRIVE3_5** for 3.5" drives and **DRIVE5_25** for 5.25" drives. The unit can be opened only if the device understands the drive type it is connected to.

DETERMINING THE NUMBER OF TRACKS OF A DRIVE

You determine the number of a tracks of a drive by passing an **IOExtTD** to the device with TD_GETNUMTRACKS set in **io_Command**. For quick I/O, you must set **io_Flags** to IOF_QUICK.

```
DiskIO->iotd_Req.io_Flags = IOF_QUICK;
DiskIO->iotd_Req.io_Command = TD_GETNUMTRACKS;
BeginIO((struct IORequest *)DiskIO);
```

TD_GETNUMTRACKS returns the number of tracks on that device in **io_Actual**. This is the number of tracks of TD_SECTOR * NUMSECS size. It is not the number of cylinders. With two heads, the number of cylinders is half of the number of tracks. The number of cylinders is equal to the number of tracks divided by the number of heads (surfaces). The standard 3.5" Amiga drive has two heads

TD_GETGEOMETRY is the preferred over TD_GETNUMTRACKS for V36 and higher versions of the operating system especially since new drive types may have more sectors or different sector sizes, etc., than standard Amiga drives.

DETERMINING THE CURRENT DISKCHANGE NUMBER

You determine the current diskchange number of a disk by passing an **IOExtTD** to the device with TD_CHANGENUM set in **io_Command**. For quick I/O, you must set **io_Flags** to IOF_QUICK.

```
DiskIO->iotd_Req.io_Flags = IOF_QUICK;
DiskIO->iotd_Req.io_Command = TD_CHANGENUM;
BeginIO((struct IORequest *)DiskIO);
```

TD_CHANGENUM returns the current value of the diskchange counter (as used by the enhanced commands) in **io_Actual**. The disk change counter is incremented each time the disk is inserted or removed.

```
ULONG change_count;

DiskIO->iotd_Req.io_Flags = IOF_QUICK;
DiskIO->iotd_Req.io_Command = TD_CHANGENUM;
BeginIO((struct IORequest *)DiskIO);
change_count = DiskIO->iotd_Req.io_Actual;   /* store current diskchange value */

DiskIO->iotd_Req.io_Length = 1;          /* Turn on the drive motor */
DiskIO->iotd_Count = change_count;
DiskIO->iotd_Req.io_Command = ETD_MOTOR;
DoIO((struct IORequest *)DiskIO);
```

Commands for Diagnostics and Repair

The trackdisk device provides commands to move the drive heads to a specific track. These commands are provided for internal diagnostics, disk repair, and head cleaning only.

MOVING THE DRIVE HEAD TO A SPECIFIC TRACK

You move the drive head to a specific track by passing an **IOExtTD** to the device with TD_SEEK or ETD_SEEK set in **io_Command**, and **io_Offset** set to the byte offset of the track to which the seek is to occur.

```
DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track);
DiskIO->iotd_Req.io_Command = TD_SEEK;
DoIO((struct IORequest *)DiskIO);
```

> *Seeking is not Reading.* TD_SEEK and ETD_SEEK do not verify their position until the next read. That is, they only move the heads; they do not actually read any data.

Notification of Disk Changes

Many programs will wish to be notified if the user has changed the disk in the active drive. While this can be done via the Intuition DISKREMOVED and DISKINSERTED messages, sometimes more tightly controlled testing is required. The trackdisk device provides commands to initiate interrupt processing when disks change.

ADDING A DISKCHANGE SOFTWARE INTERRUPT HANDLER

The trackdisk device lets you add a software interrupt handler that will be **Cause()**'ed when a disk insert or remove occurs. Within the handler, you may only call the status commands that can use IOF_QUICK.

You add a software interrupt handler by passing an **IOExtTD** to the device with a pointer to an **Interrupt** structure set in **io_Data**, the length of the structure set in **io_Length** and TD_ADDCHANGEINT set in **io_Command**.

```
DiskIO->iotd_Req.io_Length = sizeof(struct Interrupt)
DiskIO->iotd_Req.io_Data   = (APTR)Disk_Interrupt;
DiskIO->iotd_Req.io_Command = TD_ADDCHANGEINT;
SendIO((struct IORequest *)DiskIO);
```

> *Going, going, gone.* This command does *not* return when executed. It holds onto the IORequest until the **TD_REMCHANGEINT** command is executed with that same IORequest. Hence, you must use **SendIO()** with this command.

REMOVING A DISKCHANGE SOFTWARE INTERRUPT HANDLER

You remove a software interrupt handler by passing an **IOExtTD** to the device with a pointer to an **Interrupt** structure set in **io_Data**, the length of the structure set in **io_Length** and TD_REMCHANGEINT set in **io_Command**. You *must* pass it the same Interrupt structure used to add the handler.

```
DiskIO->iotd_Req.io_Length = sizeof(struct Interrupt)
DiskIO->iotd_Req.io_Data   = (APTR)Disk_Interrupt;
DiskIO->iotd_Req.io_Command = TD_REMCHANGEINT;
DoIO((struct IORequest *)DiskIO);
```

> *Don't use with pre-V36 and earlier versions.* Under pre-V36 and earlier versions of the Amiga system software, TD_REMCHANGEINT does not work and should not be used. Instead, use the workaround listed in the "trackdisk.doc" of the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Commands for Low-Level Access

The trackdisk device provides commands to read and write raw flux changes on the disk. The data returned from a low-level read or sent via a low-level write should be encoded into some form of legal flux patterns. See the *Amiga Hardware Reference Manual* and books on magnetic media recording and reading.

> *Proceed at your own risk with V1.3 and earlier versions.* In V1.3 Kickstart and earlier these functions are unreliable even though under certain configurations the commands may appear to work.

READING RAW DATA FROM A DISK

ETD_RAWREAD and TD_RAWREAD perform a raw read from a track on the disk. They seek to the specified track and read it into the user's buffer.

No processing of the track is done. It will appear exactly as the bits come off the disk – typically in some legal flux format (such as MFM, FM, GCR, etc; if you don't know what these are, you shouldn't be using this call). Caveat programmer.

This interface is intended for sophisticated programming only. You must fully understand digital magnetic recording to be able to utilize this call. It is also important that you understand that the MFM encoding scheme used by the higher level trackdisk routines may change without notice. Thus, this routine is only really useful for reading and decoding other disks such as MS-DOS formatted disks.

You read raw data from a disk by passing an **IOExtTD** to the device with TD_RAWREAD or ETD_RAWREAD set in **io_Command**, the number of bytes to be read set in **io_Length** (maximum 32K), a pointer to the read buffer set in **io_Data**, and **io_Offset** set to the byte offset of the track where you want to the read to begin. For ETD_RAWREAD, you must also set **iotd_Count** to the current diskchange number.

```
DiskIO->iotd_Req.io_Length = 1024;          /* number of bytes to read */
DiskIO->iotd_Req.io_Data = (APTR)Readbuffer; /* pointer to buffer */
DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track); /* track number */
DiskIO->iotd_Req.io_Flags = IOTDF_INDEX      /* Set for index sync */
DiskIO->iotd_Count = change_count;          /* diskchange number */
DiskIO->iotd_Req.io_Command = ETD_RAWREAD;
DoIO((struct IORequest *)DiskIO);
```

A raw read may be synched with the index pulse by setting the IOTDF_INDEXSYNC flag or synched with a $4489 sync pattern by setting the IOTDF_WORDSYNC flag. See the "trackdisk.doc" of the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for more information about these flags.

> *Forewarned is Forearmed.* Commodore-Amiga may make enhancements to the disk format in the future. Commodore-Amiga intends to provide compatibility within the trackdisk device. Anyone who uses these raw routines is bypassing this upward-compatibility and does so at her own risk.

WRITING RAW DATA TO A DISK

ETD_RAWWRITE and TD_RAWWRITE perform a raw write to a track on the disk. They seek to the specified track and write it from the user's buffer.

No processing of the track is done. It will be written exactly as the bits come out of the buffer – typically in some legal flux format (such as MFM, FM, GCR; if you don't know what these are, you shouldn't be using this call). Caveat Programmer.

This interface is intended for sophisticated programming only. You must fully understand digital magnetic recording to be able to utilize this call. It is also important that you understand that the MFM encoding scheme used by the higher level trackdisk routines may change without notice. Thus, this routine is only really useful for encoding and writing other disk formats such as MS-DOS disks.

You write raw data to a disk by passing an **IOExtTD** to the device with TD_RAWRITE or ETD_RAWRITE set in **io_Command**, the number of bytes to be written set in **io_Length** (maximum 32K), a pointer to the write buffer set in **io_Data**, and **io_Offset** set to the byte offset of the track where you want to the write to begin. For ETD_RAWWRITE, you must also set **iotd_Count** to the current diskchange number.

```
DiskIO->iotd_Req.io_Length = 1024;           /* number of bytes to write */
DiskIO->iotd_Req.io_Data = (APTR)Writebuffer; /* pointer to buffer */
DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track); /* track number */
DiskIO->iotd_Req.io_Flags = IOTDF_INDEX      /* Set for index sync */
DiskIO->iotd_Count = change_count;          /* diskchange number */
DiskIO->iotd_Req.io_Command = ETD_RAWWRITE;
DoIO((struct IORequest *)DiskIO);
```

A raw read may be synched with the index pulse by setting the IOTDF_INDEXSYNC flag or synched with a $4489 sync pattern by setting the IOTDF_WORDSYNC flag. See the "trackdisk.doc" of the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for more information about these flags.

LIMITATIONS FOR SYNC'ED READS AND WRITES

There is a delay between the index pulse and the start of bits coming in from the drive (e.g. dma started). It is in the range of 135-200 microseconds. This delay breaks down as follows: 55 microseconds for software interrupt overhead (this is the time from interrupt to the write of the DSKLEN register); 66 microsecs for one horizontal line delay (remember that disk I/O is synchronized with Agnus' display fetches). The last variable (0-65 microseconds) is an additional scan line since DSKLEN is poked anywhere in the horizontal line. This leaves 15 microseconds unaccounted for. In short, you will almost never get bits within the first 135 microseconds of the index pulse, and may not get it until 200 microseconds. At 4 microsecs/bit, this works out to be between 4 and 7 bytes of user data delay.

> *Forewarned is Forearmed.* Commodore-Amiga may make enhancements to the disk format in the future. Commodore-Amiga intends to provide compatibility within the trackdisk device. Anyone who uses these raw routines is bypassing this upward-compatibility and does so at her own risk.

Trackdisk Device Errors

The trackdisk device returns error codes whenever an operation is attempted.

```
DiskIO->iotd_Req.io_Length = TRACK_SIZE;
DiskIO->iotd_Req.io_Data = (APTR)Writebuffer;
DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * tracknum);
DiskIO->iotd_Count = change_count;
DiskIO->iotd_Req.io_Command = ETD_WRITE;
if (DoIO((struct IORequest *)DiskIO))
    printf("ETD_WRITE failed.  Error: %ld\n",DiskIO-iotd.io_Error);
```

When an error occurs, these error numbers will be returned in the **io_Error** field of your **IOExtTD** block.

Trackdisk Device Error Codes

Error	**Value**	**Explanation**
TDERR_NotSpecified	20	Error could not be determined
TDERR_NoSecHdr	21	Could not find sector header
TDERR_BadSecPreamble	22	Error in sector preamble
TDERR_BadSecID	23	Error in sector identifier
TDERR_BadHdrSum	24	Header field has bad checksum
TDERR_BadSecSum	25	Sector data field has bad checksum
TDERR_TooFewSecs	26	Incorrect number of sectors on track
TDERR_BadSecHdr	27	Unable to read sector header
TDERR_WriteProt	28	Disk is write-protected
TDERR_DiskChanged	29	Disk has been changed or is not currently present
TDERR_SeekError	30	While verifying seek position, found seek error
TDERR_NoMem	31	Not enough memory to do this operation
TDERR_BadUnitNum	32	Bad unit number (unit # not attached)
TDERR_BadDriveType	33	Bad drive type (not an Amiga 3 1/2 inch disk)
TDERR_DriveInUse	34	Drive already in use (only one task exclusive)
TDERR_PostReset	35	User hit reset; awaiting doom

Example Trackdisk Program

```
/*
 * Track_Copy.c
 *
 * This program does a track by track copy from one drive to another
 *
 * Compile with SAS C 5.10  LC -cfist -ms -v -L
 *
 * This program will only run from the CLI.  If started from
 * the workbench, it will just exit...
 *
 * Usage:  trackcopy  dfx dfy
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <devices/trackdisk.h>
#include <dos/dosextens.h> */

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>

#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

#define TRACK_SIZE      ((LONG)(NUMSECS * TD_SECTOR))


/*
 * Turn the BUSY flag off/on for the drive
 * If onflag is TRUE, the disk will be marked as busy...
 *
 * This is to stop the validator from executing while
 * we are playing with the disks.
 */
VOID disk_busy(UBYTE *drive,LONG onflag)
{
struct StandardPacket *pk;
struct Process        *tsk;

    tsk=(struct Process *)FindTask(NULL);
    if (pk=AllocMem(sizeof(struct StandardPacket),MEMF_PUBLIC|MEMF_CLEAR))
    {
        pk->sp_Msg.mn_Node.ln_Name=(UBYTE *)&(pk->sp_Pkt);

        pk->sp_Pkt.dp_Link=&(pk->sp_Msg);
        pk->sp_Pkt.dp_Port=&(tsk->pr_MsgPort);
        pk->sp_Pkt.dp_Type=ACTION_INHIBIT;
        pk->sp_Pkt.dp_Arg1=(onflag ? -1L : 0L);

        PutMsg(DeviceProc(drive),(struct Message *)pk);
        WaitPort(&(tsk->pr_MsgPort));
        GetMsg(&(tsk->pr_MsgPort));
        FreeMem(pk,(long)sizeof(*pk));
    }
}


/*
 * This turns the motor off
 */
VOID Motor_Off(struct IOExtTD *disk)
{
    disk->iotd_Req.io_Length=0;
    disk->iotd_Req.io_Command=TD_MOTOR;
    DoIO((struct IORequest *)disk);
}
```

```
/*
 * This turns the motor on
 */
VOID    Motor_On(struct IOExtTD *disk)
{
    disk->iotd_Req.io_Length=1;
    disk->iotd_Req.io_Command=TD_MOTOR;
    DoIO((struct IORequest *)disk);
}


/*
 * This reads a track, reporting any errors...
 */

SHORT Read_Track(struct IOExtTD *disk,UBYTE *buffer,SHORT track)
{
SHORT All_OK=TRUE;

    disk->iotd_Req.io_Length=TRACK_SIZE;
    disk->iotd_Req.io_Data=(APTR)buffer;
    disk->iotd_Req.io_Command=CMD_READ;
    disk->iotd_Req.io_Offset=(ULONG)(TRACK_SIZE * track);
    DoIO((struct IORequest *)disk);
    if (disk->iotd_Req.io_Error)
    {
        All_OK=FALSE;
        printf("Error %u when reading track %d",disk->iotd_Req.io_Error,track);
    }
    return(All_OK);
}


/*
 * This writes a track, reporting any errors...
 */

SHORT Write_Track(struct IOExtTD *disk,UBYTE *buffer,SHORT track)
{
SHORT All_OK=TRUE;

    disk->iotd_Req.io_Length=TRACK_SIZE;
    disk->iotd_Req.io_Data=(APTR)buffer;
    disk->iotd_Req.io_Command=TD_FORMAT;
    disk->iotd_Req.io_Offset=(ULONG)(TRACK_SIZE * track);
    DoIO((struct IORequest *)disk);
    if (disk->iotd_Req.io_Error)
    {
        All_OK=FALSE;
        printf("Error %d when writing track %d",disk->iotd_Req.io_Error,track);
    }
    return(All_OK);
}


/*
 * This function finds the number of TRACKS on the device.
 * NOTE That this is TRACKS and not cylinders.  On a Two-Head
 * drive (such as the standard 3.5" drives) the number of tracks
 * is 160, 80 cylinders, 2-heads.
 */

SHORT FindNumTracks(struct IOExtTD *disk)
{
    disk->iotd_Req.io_Command=TD_GETNUMTRACKS;
    DoIO((struct IORequest *)disk);
    return((SHORT)disk->iotd_Req.io_Actual);
}
```

```
/*
 * This routine allocates the memory for one track and does
 * the copy loop.
 */

VOID Do_Copy(struct IOExtTD *diskreq0,struct IOExtTD *diskreq1)
{
UBYTE *buffer;
SHORT track;
SHORT All_OK;
SHORT NumTracks;

    if (buffer=AllocMem(TRACK_SIZE,MEMF_CHIP|MEMF_PUBLIC))
    {
        printf(" Starting Motors\r");
        Motor_On(diskreq0);
        Motor_On(diskreq1);
        All_OK=TRUE;

        NumTracks=FindNumTracks(diskreq0);

        for (track=0;(track<NumTracks) && All_OK;track++)
        {
            printf(" Reading track %d\r",track);

            if (All_OK=Read_Track(diskreq0,buffer,track))
            {
                printf(" Writing track %d\r",track);

                All_OK=Write_Track(diskreq1,buffer,track);
            }
        }
        if (All_OK) printf(" * Copy complete *");
        printf("\n");
        Motor_Off(diskreq0);
        Motor_Off(diskreq1);
        FreeMem(buffer,TRACK_SIZE);
    }
    else printf("No memory for track buffer...\n");
}


/*
 * Prompts the user to remove one of the disks.
 * Since this program makes an EXACT copy of the disks
 * AmigaDOS would get confused by them so one must be removed
 * before the validator is let loose.  Also, note that the
 * disks may NEVER be in drives on the SAME computer at the
 * SAME time unless one of the disks is renamed.  This is due
 * to a bug in the system.  It would normally be prevented
 * by a diskcopy program that knew the disk format and modified
 * the creation date by one clock-tick such that the disks would
 * be different.
 */

VOID Remove_Disks(VOID)
{
    printf("\nYou *MUST* remove at least one of the disks now.\n");
    printf("\nPress RETURN when ready\n");
    while(getchar()!='\n');
}


/*
 * Prompts the user to insert the disks.
 */

VOID Insert_Disks(char drive1[], char drive2[])
{
    printf("\nPlease insert source disk in %s\n",drive1);
    printf("\n          and destination in %s\n",drive2);
    printf("\nPress RETURN when ready\n");
    while(getchar()!='\n');
}
```

```
/*
 * Open the devices and mark them as busy
 */
VOID Do_OpenDevice(struct IOExtTD *diskreq0,struct IOExtTD *diskreq1, long unit[])
{
char drive1[] = "DFx:";  /* String for source drive */
char drive2[] = "DFx:";  /* String for destination drive */

    drive1[2] = unit[0]+ '0';  /* Set drive number for source */

    if (!OpenDevice(TD_NAME,unit[0],(struct IORequest *)diskreq0,0L))
    {
          disk_busy(drive1,TRUE);
          drive2[2] = unit[1]+ '0';  /* Set drive number for destination */

        if (!OpenDevice(TD_NAME,unit[1],(struct IORequest *)diskreq1,0L))
        {
            disk_busy(drive2,TRUE);

            Insert_Disks(drive1,drive2);
            Do_Copy(diskreq0,diskreq1);
            Remove_Disks();

            disk_busy(drive2,FALSE);
            CloseDevice((struct IORequest *)diskreq1);
        }
        else printf("Could not open %s\n",drive2);

        disk_busy(drive1,FALSE);
        CloseDevice((struct IORequest *)diskreq0);
    }
    else printf("Could not open %s\n",drive1);
}


SHORT ParseArgs(int argc, char **argv, long Unit[])
#define OKAY 1
{
int j=1, params = OKAY;
char *position[]={"First","Second"};

if (argc != 3)
    {
    printf("\nYou must specify a source and destination disk\n");
    return(!OKAY);
    }
else if (strcmp(argv[1],argv[2]) == 0)
          {
          printf("\nYou must specify different disks for source and destination\n");
          return(!OKAY);
          }
     else while (params == OKAY && j<3)
          {
          if (strnicmp(argv[j],"df",2)==0)
            {
            if (argv[j][2] >= '0' && argv[j][2] <= '3' && argv[j][3] == '\0')
              {
              Unit[j-1] = argv[j][2] - 0x30;
              }
            else
              {
              printf("\n%s parameter is wrong, unit number must be 0-3\n",position[j-1]);
              params = !OKAY;
              return(!OKAY);
              }
            }
          else
            {
            printf("\n%s parameter is wrong, you must specify a floppy device df0 - df3\n",
                    position[j-1]);
            params=!OKAY;
            return(!OKAY);
            }
          j++;
          }
return(OKAY);
}
```

```
VOID main(int argc,char **argv)
{
struct IOExtTD *diskreq0;
struct IOExtTD *diskreq1;
struct MsgPort *diskPort;
long unit[2];

    if (ParseArgs(argc, argv, unit))       /* Check inputs */
    {
        if (diskPort=CreatePort(NULL,NULL))
        {
            if (diskreq0=(struct IOExtTD *)CreateExtIO(diskPort,
                                                sizeof(struct IOExtTD)))
            {
                if (diskreq1=(struct IOExtTD *)CreateExtIO(diskPort,
                                                sizeof(struct IOExtTD)))
                {
                    Do_OpenDevice(diskreq0,diskreq1, unit);
                    DeleteExtIO((struct IORequest *)diskreq1);
                }
                else printf("Out of memory\n");
                DeleteExtIO((struct IORequest *)diskreq0);
            }
            else printf("Out of memory\n");
            DeletePort(diskPort);
        }
        else printf("Could not create diskReq port\n");
    }
}
```

Only one per customer. Since this example program makes an exact track-for-track duplicate, AmigaDOS will get confused if both disks are in drives on the system at the same time. While the disks are inhibited, this does not cause a problem, but during normal operation, this will cause a system hang. To prevent this, you can relabel one of the disks. A commercial diskcopy program would have to understand the disk format and either relabel the disk or modify the volume creation date/time by a bit in order to make the disks look different to the system.

Additional Information on the Trackdisk Device

Additional programming information on the trackdisk device can be found in the include files and the autodocs for the trackdisk device. Both are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Trackdisk Device Information	
INCLUDES	devices/trackdisk.h devices/trackdisk.i
AUTODOCS	trackdisk.doc

chapter fifteen
RESOURCES

The Amiga's low-level hardware control functions are collectively referred to as "Resources". Most applications will never need to use the hardware at the resource level—the Amiga's device interface is much more convenient and provides for multitasking. However, some high performance applications, such as MIDI time stamping, may require direct access to the Amiga hardware registers.

New Features for Version 2.0	
Feature	**Description**
BattClock	New resource
BattMem	New resource
FileSystem	New resource

Compatibility Warning: The new features for 2.0 are not backwards compatible.

The Amiga Resources

There are currently seven standard resources in the Amiga system. The following lists the name of each resource and its function.

battclock.resource
grants access to the battery-backed clock chip.

battmem.resource
grants access to non-volatile RAM.

cia.resource
grants access to the interrupts and timer bits of the 8520 CIA (Complex Interface Adapter) chips.

disk.resource
grants temporary exclusive access to the disk hardware.

FileSystem.resource
grants access to the file system.

misc.resource
grants exclusive access to functional blocks of chip registers. At present, definitions have been made for the serial and parallel hardware only.

potgo.resource
manages the bits of the proportional I/O pins on the game controller ports.

The resources allow you direct access to the hardware in a way that is compatible with multitasking. They also allow you to temporarily bar other tasks from using the resource. You may then use the associated hardware directly for your special purposes. If applicable, you must return the resource back to the system for other tasks to use when you are finished with it.

See the *Amiga Hardware Reference Manual* for detailed information on the actual hardware involved.

> *Look Before You Leap.* Resources are just one step above direct hardware manipulation. You are advised to try the higher level device and library approach before resorting to the hardware.

Resource Interface

Resources provide functions that you call to do low-level operations with the hardware they access. In order to use the functions of a resource, you must obtain a pointer to the resource. This is done by calling the **OpenResource()** function with the resource name as its argument.

OpenResource() returns a pointer to the resource you request or NULL if it does not exist.

```
#include <resources/filesysres.h>

struct FileSysResource *FileSysResBase = NULL;

if (!(FileSysResBase = OpenResource(FSRNAME)))
    printf("Cannot open %s\n",FSRNAME);
```

There is no **CloseResource()** function. When you are done with a resource, you are done with it. However, as you will see later in this chapter, some resources provide functions to allocate parts of the hardware they access. In those cases, you will have to free those parts for anyone else to use them.

Each resource has at least one include file in the *resources* subdirectory of the include directory. Some of the include files contain only the name of the resource; others list structures and bit definitions used by the resource. The include files will be listed at the end of this chapter.

Calling a resource function is the same as calling any other function on the Amiga. You have to know what parameters it accepts and the return value, if any. The Autodocs for each resource lists the functions and their requirements.

```
#include <hardware/cia.h>
#include <resources/cia.h>

struct Library *CIAResource = NULL;

void main()
{

WORD mask = 0;

if (!(CIAResource = OpenResource(CIABNAME)))
    printf("Cannot open %s\n",CIABNAME);
else
   {
    /* What is the interrupt enable mask? */
     mask = AbleICR(CIAResource,0);

    printf("\nThe CIA interrupt enable mask: %x \n",mask);
   }
}
```

Looks Can Be Deceiving. Some resources may look like libraries and act like libraries, but be assured they are not libraries.

BattClock Resource

The battery-backed clock (BattClock) keeps Amiga time while the system is powered off. The time from the BattClock is loaded into the Amiga system clock as part of the boot sequence.

The battclock resource provides access to the BattClock. Three functions allow you to read the BattClock value, reset it and set it to a value you desire.

BattClock Resource Functions

ReadBattClock()	Read the time from the BattClock and returns it as the number of seconds since 12:00 AM, January 1, 1978.
ResetBattClock()	Reset the BattClock to 12:00 AM, January 1, 1978.
WriteBattClock()	Set the BattClock to the number of seconds you pass it relative to 12:00 AM, January 1, 1978.

The *utility.library* contains time functions which convert the number of seconds since 12:00 AM, January 1, 1978 to a date and time we can understand, and vice versa. You will find these functions useful when dealing with the BattClock. The example program below uses the **Amiga2Date()** utility function to convert the value returned by **ReadBattClock()**. See the "Utility Library" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for a discussion of the *utility.library* and the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for a listing of its functions.

So, You Want to Be A Time Lord? This resource will allow you to set the BattClock to any value you desire. Keep in mind that this time will endure a reboot and could adversely affect your system.

```
/*
 * Read_BattClock.c
 *
 * Example of reading the BattClock and converting its output to
 * a useful measure of time by calling the Amiga2Date() utility function.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <dos/dos.h>
#include <utility/date.h>
#include <resources/battclock.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/battclock_protos.h>
#include <clib/utility_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

VOID main(VOID);

struct Library *UtilityBase = NULL;
struct Library *BattClockBase;

VOID main(VOID)
{
UBYTE *Days[] ={"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"};
UBYTE *Months[] = {"January","February","March","April","May","June",
                   "July","August","September","October","November","December"};
UBYTE *ampm;
ULONG AmigaTime;
struct ClockData MyClock;

if (UtilityBase = (struct Library *)OpenLibrary("utility.library",33))
    {
    if (BattClockBase= OpenResource(BATTCLOCKNAME))
        {
        /* Get number of seconds till now */
        AmigaTime = ReadBattClock();

        /* Convert to a ClockData structure */
        Amiga2Date(AmigaTime,&MyClock);

        printf("\nRobin, tell everyone the BatDate and BatTime");

        /* Print the Date */
        printf("\n\nOkay Batman, the BatDate is ");
        printf("%s, %s %d, %d",Days[MyClock.wday],Months[MyClock.month-1],
                               MyClock.mday,MyClock.year);

        /* Convert military time to normal time and set AM/PM */
        if (MyClock.hour < 12)
            ampm = "AM";        /* hour less than 12, must be morning */
        else
            {
            ampm = "PM";         /* hour greater than 12,must be night */
            MyClock.hour -= 12;  /* subtract the extra 12 of military */
            };
        if (MyClock.hour == 0)
            MyClock.hour = 12;   /* don't forget the 12s */

        /* Print the time */
        printf("\n          the BatTime is ");
        printf("%d:%02d:%02d %s\n\n",MyClock.hour,MyClock.min,MyClock.sec,ampm);
        }
    else
       printf("Error: Unable to open the %s\n",BATTCLOCKNAME);

    /* Close the utility library */
    CloseLibrary(UtilityBase);
    }
else
    printf("Error: Unable to open utility.library\n");
}
```

Additional programming information on the battclock resource can be found in the include files and the Autodocs for the battclock resource and the utility library.

BattMem Resource

The battery-backed memory (BattMem) preserves a small portion of Amiga memory while the system is powered off. Some of the information stored in this memory is used during the system boot sequence.

The battmem resource provides access to the BattMem. Four functions allow you to use the BattMem.

BattMem Resource Functions

ObtainBattSemaphore	Obtain exclusive access to the BattMem.
ReadBattMem()	Read a bitstring from the BattMem. You specify the bit position and the number of bits you wish to read.
ReleaseBattSemaphore()	Relinquish exclusive access to the BattMem.
WriteBattMem()	Write a bitstring to the BattMem. You specify the bit position and the number of bits you wish to write.

The system considers BattMem to be a set of bits rather than bytes. This is done to conserve the limited space available. All bits are reserved, and applications should not read, or write undefined bits. Writing bits should be done with extreme caution since the settings will survive power-down/power-up. You can find the bit definitions in the BattMem include files *resources/battmembitsamiga.h*, *resources/battmembitsamix.h* and *resources/battmembitsshared.h*. They should be consulted before you do anything with the resource.

You Don't Need This Resource. The BattMem resource is basically for system use only. There is generally no reason for applications to use it. It is documented here simply for completeness.

Additional information on the battmem resource can be found in the include files and the Autodocs for the battmem resource.

BattMem Resource Information	
INCLUDES	resources/battmem.i resources/battmembitsamiga.h resources/battmembitsamix.h resources/battmembitsshared.h
AUTODOCS	battmem.doc

CIA Resource

The CIA resource provides access to the timers and timer interrupt bits of the 8520 Complex Interface Adapter (CIA) A and B chips. This resource is intended for use by high performance timing applications such as MIDI time stamping and SMPTE time coding.

Four functions allow you to interact with the CIA hardware.

CIA Resource Functions

AbleICR()	Enable or disable Interrupt Control Register interrupts. Can also return the current or previous enabled interrupt mask.
AddICRVector()	Allocate one of the CIA timers by assigning an interrupt handler to an interrupt bit and enabling the interrupt of one of the timers. If the timer you request is not available, a pointer to the interrupt structure that owns it will be returned.
RemICRVector()	Remove an interrupt handler from an interrupt bit and disable the interrupt.
SetICR()	Cause or clear one or more interrupts, or return the current or previous interrupt status.

Each CIA chip has two interval timers within it—Timer A and Timer B—that may be available. The CIA chips operate at different interrupt levels with the CIA-A timers at interrupt level 2 and the CIA-B timers at interrupt level 6.

> *Choose A Timer Wisely.* The timer you use should be based solely on interrupt level and availability. If the timer you request is not available, try for another. Whatever you do, do not base your decision on what you think the timer is used for by the system.

You allocate a timer by calling **AddICRVector()**. This is the only way you should access a timer. If the function returns zero, you have successfully allocated that timer. If it is unavailable, the owner interrupt will be returned.

```
/* allocate CIA-A Timer A */
inta = AddICRVector (CIAResource, CIAICRB_TA, &tint);

if (inta)  /* if allocate was not successful */
    printf("Error: Could not allocate timer\n");
else
    {
    ...ready for timing
    }
```

The timer is deallocated by calling **RemICRVector()**. This is the only way you should deallocate a timer.

```
RemICRVector(CIAResource, CIAICRB_TA, &tint);
```

Your application should not make any assumptions regarding which interval timers (if any) are available for use; other tasks or critical operating system routines may be using the interval timers. In fact, in the latest version of the operating system, the timer device may dynamically allocate one of the interval timers.

> *Time Is Of The Essence!* There are a limited number of free CIA interval timers. Applications which use the interval timers may not be able to run at the same time if all interval timers are in use. As a general rule, you should use the timer device for most interval timing.

You read from and write to the CIA interrupt control registers using **SetICR()** and **AbleICR()**. **SetICR()** is useful for sampling which cia interrupts (if any) are active. It can also be used to clear and generate interrupts. **AbleICR()** is used to disable and enable a particular CIA interrupt. Additional information about these functions can be found in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

Things to keep in mind:

1. *Never* directly read from or write to the CIA interrupt control registers. Always use **SetICR()** and **AbleICR()**.
2. Your interrupt routine will be called with a pointer to your data area in register A1, and a pointer to the code being called in register A5. No other registers are set up for you. You must observe the standard convention of preserving all registers except D0–D1 and A0–A1.
3. Never turn off all level 2 or level 6 interrupts. The proper way to disable interrupts for an interval timer that you've successfully allocated is via the **AbleICR()** function.
4. Interrupt handling code should be written in assembly code and, if possible, should signal a task to do most of the work.
5. Do not make assumptions about which CIA interval timers (if any) are available for use. The only proper way to own an interval timer is via the **AddICRVector()** function.
6. Do not use **SetICR()**, **AbleICR()** and **RemICRVector()** to affect timers or other CIA hardware which your task does not own.

Changes in the CIA resource:

- In pre-V36 versions of the operating system, **SetICR()** could return FALSE for a particular interrupt just prior to processing the interrupt. **SetICR()** now returns TRUE for a particular interrupt until sometime after the interrupt has been processed.
- Applications which only need to read a CIA interval timer should use the **ReadEClock()** function of the timer device. See the "Timer Device" chapter of this manual for more information on **ReadEClock()**.
- The timer device may dynamically allocate a free CIA interval timer. Do not make any assumptions regarding which interval timers are in use unless you are taking over the machine *completely*.

```
/*
 * Cia_Interval.c
 *
 * Demonstrate allocation and use of a cia interval timer
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/tasks.h>
#include <exec/interrupts.h>
#include <hardware/cia.h>
#include <resources/cia.h>

#include <clib/exec_protos.h>
#include <clib/cia_protos.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>


/* prototypes */

void    StartTimer      (struct freetimer *ft, struct exampledata *ed);
int     FindFreeTimer   (struct freetimer *ft, int preferA);
int     TryTimer        (struct freetimer *ft);
void    main            ( USHORT, char **);


/* see usage of these defines in StartTimer() below */

#define COUNTDOWN 20
#define HICOUNT 0xFF
#define LOCOUNT 0xFF

#define STOPA_AND  CIACRAF_TODIN |CIACRAF_PBON | CIACRAF_OUTMODE | CIACRAF_SPMODE

        /*
        ;
        ; AND mask for use with control register A
        ; (interval timer A on either CIA)
        ;
        ; STOP -
        ;       START bit 0 == 0 (STOP IMMEDIATELY)
        ;       PBON  bit 1 == same
        ;       OUT   bit 2 == same
        ;       RUN   bit 3 == 0 (SET CONTINUOUS MODE)
        ;       LOAD  bit 4 == 0 (NO FORCE LOAD)
        ;       IN    bit 5 == 0 (COUNTS 02 PULSES)
        ;       SP    bit 6 == same
        ;       TODIN bit 7 == same (unused on ciacra)

        */

#define STOPB_AND  CIACRBF_ALARM | CIACRBF_PBON | CIACRBF_OUTMODE

        /*
        ;
        ; AND mask for use with control register B
        ; (interval timer B on either CIA)
        ;
        ; STOP -
        ;       START bit 0 == 0 (STOP IMMEDIATELY)
        ;       PBON  bit 1 == same
        ;       OUT   bit 2 == same
        ;       RUN   bit 3 == 0 (SET CONTINUOUS MODE)
        ;       LOAD  bit 4 == 0 (NO FORCE LOAD)
        ;       IN0   bit 5 == 0 (COUNTS 02 PULSES)
        ;       IN1   bit 6 == 0 (COUNTS 02 PULSES)
        ;       ALARM bit 7 == same (TOD alarm control bit)

        */
```

```
#define STARTA_OR  CIACRAF_START

        /*
        ;
        ; OR mask for use with control register A
        ; (interval timer A on either CIA)
        ;
        ; START -
        ;
        ;      START bit 0 == 1 (START TIMER)
        ;
        ;      All other bits unaffected.
        ;

        */

#define STARTB_OR  CIACRBF_START

        /*
        ;
        ; OR mask for use with control register B
        ; (interval timer A on either CIA)
        ;
        ; START -
        ;
        ;      START bit 0 == 1 (START TIMER)
        ;
        ;      All other bits unaffected.
        ;

        */


/*
 * Structure which will be used to hold all relevant information about
 * the cia timer we manage to allocate.
 *
 */

struct freetimer
{
    struct Library *ciabase;        /* CIA Library Base            */
    ULONG  timerbit;                /* timer bit allocated         */
    struct CIA *cia;                /* ptr to hardware             */
    UBYTE *ciacr;                   /* ptr to control register     */
    UBYTE *cialo;                   /* ptr to low byte of timer    */
    UBYTE *ciahi;                   /* ptr to high byte of timer   */
    struct Interrupt timerint;      /* Interrupt structure         */
    UBYTE  stopmask;                /* Stop/set-up timer           */
    UBYTE  startmask;               /* Start timer                 */
};

/*
 * Structure which will be used by the interrupt routine called
 * when our cia interval timer generates an interrupt.
 *
 */

struct exampledata
{
    struct Task *task;      /* task to signal */
    ULONG   signal;         /* Signal bit to use */
    ULONG   counter;
};


struct CIA *ciaa = (struct CIA *)0xbfe001;
struct CIA *ciab = (struct CIA *)0xbfd000;


#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

/*
```

```
 * This is the interrupt routine which will be called when our CIA
 * interval timer counts down.
 *
 * This example decrements a counter each time the interrupt routine
 * is called until the counter reaches 0, at which time it signals
 * our main task.
 *
 * Note that interrupt handling code should be efficient, and will
 * generally be written in assembly code.  Signaling another task
 * such as this example does is also a useful way of handling
 * interrupts in an expedient manner.
 */

void __asm ExampleInterrupt(register __a1 struct exampledata *ed)
{
if (ed->counter)
    {
    ed->counter--;                  /* decrement counter */
    }
else
    {
    ed->counter = COUNTDOWN;        /* reset counter     */

    Signal(ed->task,(1L << ed->signal));
    }
}


/***********************************
 *  main()
 ***********************************/

void main(USHORT argc,char **argv)
{
struct freetimer ft;
struct exampledata ed;

/* Set up data which will be passed to interrupt */

ed.task = FindTask(0L);

if (ed.signal = AllocSignal(-1L))
    {
    /* Prepare freetimer structure : set-up interrupt */

    ft.timerint.is_Node.ln_Type = NT_INTERRUPT;
    ft.timerint.is_Node.ln_Pri  = 0;
    ft.timerint.is_Node.ln_Name = "cia_example";

    ft.timerint.is_Data         = (APTR)&ed;
    ft.timerint.is_Code         = (APTR)ExampleInterrupt;

    /* Call function to find a free CIA interval timer
     * with flag indicating that we prefer a CIA-A timer.
     */

    printf("Attempting to allocate a free timer\n");

    if (FindFreeTimer(&ft,TRUE))
        {
        if (ft.cia == ciaa)
            {
            printf("CIA-A timer ");
            }
        else
            {
            printf("CIA-B timer ");
            }

        if (ft.timerbit == CIAICRB_TA)
            {
            printf("A allocated\n");
            }
        else
            {
            printf("B allocated\n");
            }
        /* We found a free interval timer.  Let's start it running. */

        StartTimer(&ft,&ed);

        /* Wait for a signal */

        printf("Waiting for signal bit %ld\n",ed.signal);

        Wait(1L<<ed.signal);

        printf("We woke up!\n");

        /* Release the interval timer */

        RemICRVector(ft.ciabase,ft.timerbit,&ft.timerint);

        }
    else
        {
        printf("No CIA interval timer available\n");
        }

    FreeSignal(ed.signal);
    }
}


/*
 * This routine sets up the interval timer we allocated with
 * AddICRVector().  Note that we may have already received one, or
 * more interrupts from our timer.  Make no assumptions about the
 * initial state of any of the hardware registers we will be using.
 *
 */

void StartTimer(struct freetimer *ft, struct exampledata *ed)
{
register struct CIA *cia;

cia = ft->cia;

/* Note that there are differences between control register A,
 * and B on each CIA (e.g., the TOD alarm bit, and INMODE bits.
 */

if (ft->timerbit == CIAICRB_TA)
    {
    ft->ciacr = &cia->ciacra;         /* control register A   */
    ft->cialo = &cia->ciatalo;        /* low byte counter     */
    ft->ciahi = &cia->ciatahi;        /* high byte counter    */

    ft->stopmask = STOPA_AND;        /* set-up mask values   */
    ft->startmask = STARTA_OR;
    }
else
    {
    ft->ciacr = &cia->ciacrb;         /* control register B   */
    ft->cialo = &cia->ciatblo;        /* low byte counter     */
    ft->ciahi = &cia->ciatbhi;        /* high byte counter    */

    ft->stopmask = STOPB_AND;        /* set-up mask values   */
    ft->startmask = STARTB_OR;
    }


/* Modify control register within Disable().  This is done to avoid
 * race conditions since our compiler may generate code such as:
 *
 *      value = Read hardware byte
 *      AND  value with MASK
 *      Write value to hardware byte
 *
 * If we take a task switch in the middle of this sequence, two tasks
 * trying to modify the same register could trash each others' bits.
 *
 * Normally this code would be written in assembly language using atomic
 * instructions so that the Disable() would not be needed.
 */


Disable();

/* STOP timer, set 02 pulse count-down mode, set continuous mode */

*ft->ciacr &= ft->stopmask;
Enable();

/* Clear signal bit - interrupt will signal us later */
SetSignal(0L,1L<<ed->signal);

/* Count-down X # of times */
ed->counter = COUNTDOWN;

/* Start the interval timer - we will start the counter after
 * writing the low, and high byte counter values
 */

*ft->cialo = LOCOUNT;
*ft->ciahi = HICOUNT;

/* Turn on start bit - same bit for both A, and B control regs  */

Disable();
*ft->ciacr |= ft->startmask;

Enable();
}


/*
 * A routine to find a free interval timer.
 *
 * This routine makes no assumptions about which interval timers
 * (if any) are available for use.  Currently there are two interval
 * timers per CIA chip.
 *
 * Because CIA usage may change in the future, your code should use
 * a routine like this to find a free interval timer.
 *
 * Note that the routine takes a preference flag (which is used to
 * to indicate that you would prefer an interval timer on CIA-A).
 * If the flag is FALSE, it means that you would prefer an interval
 * timer on CIA-B.
 *
 */

FindFreeTimer(struct freetimer *ft, int preferA)
{
struct CIABase *ciaabase, *ciabbase;

/* get pointers to both resource bases */

ciaabase = OpenResource(CIAANAME);
ciabbase = OpenResource(CIABNAME);

/* try for a CIA-A timer first ? */

if (preferA)
    {
    ft->ciabase = ciaabase; /* library address  */
    ft->cia     = ciaa;     /* hardware address */
    }
else
    {
    ft->ciabase = ciabbase; /* library address  */
    ft->cia     = ciab;     /* hardware address */
    }

if (TryTimer(ft))
    return(TRUE);

/* try for an interval timer on the other cia */
```

```
if (!(preferA))
    {
    ft->ciabase = ciaabase; /* library address  */
    ft->cia     = ciaa;     /* hardware address */
    }
else
    {
    ft->ciabase = ciabbase; /* library address  */
    ft->cia     = ciab;     /* hardware address */
    }

if (TryTimer(ft))
    return(TRUE);

return(FALSE);

}


/*
 * Try to obtain a free interval timer on a CIA.
 */

TryTimer(struct freetimer *ft)
{

if (!(AddICRVector(ft->ciabase,CIAICRB_TA,&ft->timerint)))
    {
    ft->timerbit = CIAICRB_TA;
    return(TRUE);
    }

if (!(AddICRVector(ft->ciabase,CIAICRB_TB,&ft->timerint)))
    {
    ft->timerbit = CIAICRB_TB;
    return(TRUE);
    }

return(FALSE);
}
```

Additional programming information on the CIA resource can be found in the include files and the Autodocs for the CIA resource and the 8520 spec. The includes files and Autodocs are in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* and the 8520 spec is in the *Amiga Hardware Reference Manual.*

CIA Resource Information	
INCLUDES	resources/cia.h
	resources/cia.i
	hardware/cia.h
	hardware/cia.i
AUTODOCS	cia.doc
HARDWARE	8520 specification

Disk Resource

The Disk resource obtains exclusive access to the floppy disk hardware There are four disk/MFM units available, units 0–3.

Six functions are available for dealing with the floppy disk hardware.

Disk Resource Functions

AllocUnit()	Allocate one of the units of the disk resource.
FreeUnit()	Deallocate an allocated disk unit.
GetUnit()	Allocate the disk for a driver.
GetUnitID()	Return the drive ID of a specified drive unit.
GiveUnit()	Free the disk.
ReadUnitID()	Reread and return the drive ID of a specified unit.

The disk resource provides both a gross and a fine unit allocation scheme. **AllocUnit()** and **FreeUnit()** are used to claim a unit for long term use, and **GetUnit()** and **GiveUnit()** are used to claim a unit and the disk hardware for shorter periods.

The trackdisk device uses and abides by both allocation schemes. Because a trackdisk unit is never closed for Amiga 3.5" drives (the file system keeps them open) the associated resource units will always be allocated for these drives. **GetUnit()** and **GiveUnit()** can still be used, however, by other applications that have not succeeded with **AllocUnit()**.

You must not change the state of of a disk that the trackdisk device is using unless you either
a) force its removal before giving it up, or
b) return it to the original track (with no changes to the track), or
c) CMD_STOP the unit before **GetUnit()**, update the current track number and CMD_START it after **GiveUnit()**. This option is only available under V36 and higher versions of the operating system.

ReadUnitID() is provided to handle drives which use the unit number in a dynamic manner. Subsequent **GetUnit()** calls will return the value obtained by **ReadUnitID()**.

It is therefore possible to prevent the trackdisk device from using units that have not yet been mounted by successfully performing an **AllocUnit()** for that unit. It is also possible to starve trackdisk usage by performing a **GetUnit()**. The appropriate companion routine (**FreeUnit()** or **GiveUnit()**) should be called to restore the resource at the end of its use.

```
/*
 * Get_Disk_Unit_ID.c
 *
 * Example of getting the UnitID of a disk
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <resources/disk.h>

#include <clib/exec_protos.h>

#include <stdio.h>
```

```
#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */

/* There is no amiga.lib stub for this function so a pragma is required
 * This is a pragma for SAS C
 * Your compiler may require a different format
 */
#pragma libcall DiskBase GetUnitID 1e 1
#endif

struct Library *DiskBase = NULL;

LONG GetUnitID(long);

void main(int argc, char **argv)
{
LONG ids= 0;
LONG type;

if (!(DiskBase= (struct Library *)OpenResource(DISKNAME)))
    printf("Cannot open %s\n,DISKNAME");
else
   {
    printf("Defined drive types are:\n");
    printf("  AMIGA  $00000000\n");
    printf("  5.25''  $55555555\n");
    printf("  AMIGA  $00000000 (high density)\n");
    printf("  None   $FFFFFFFF\n\n");

    /* What are the UnitIDs? */
     for (ids = 0; ids < 4; ids++)
        {
         type = GetUnitID(ids);
         printf("The UnitID for unit %d is $%08lx\n",ids,type);
        }
   }
}
```

Additional programming information on the disk resource can be found in the include files and the Autodocs for the disk resource.

Disk Resource Information	
INCLUDES	resources/disk.h resources/disk.i
AUTODOCS	disk.doc

FileSystem Resource

The FileSystem resource returns the filesystems that are available on the Amiga. It has no functions. Opening the FileSystem resource returns a pointer to a **List** structure containing the current filesystems in the Amiga.

```
/*
 * Get_Filesys.c
 *
 * Example of examining the FileSysRes list
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <resources/filesysres.h>

#include <clib/exec_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct FileSysResource *FileSysResBase = NULL;

void main(int argc, char **argv)
{

struct FileSysEntry *fse;
int x;

/* NOTE - you should actually be in a Forbid while accessing any
 * system list for which no other method of arbitration is available.
 * However, for this example we will be printing the information
 * (which would break a Forbid anyway) so we won't Forbid.
 * In real life, you should Forbid, copy the information you need,
 * Permit, then print the info.
 */
if (!(FileSysResBase = (struct FileSysResource *)OpenResource(FSRNAME)))
    printf("Cannot open %s\n",FSRNAME);
else
    {
    for ( fse = (struct FileSysEntry *)FileSysResBase->fsr_FileSysEntries.lh_Head;
          fse->fse_Node.ln_Succ;
          fse = (struct FileSysEntry *)fse->fse_Node.ln_Succ)
         {
         printf("Found filesystem creator: %s\n",fse->fse_Node.ln_Name);

         printf("                    DosType: ");
         for (x=24; x>=8; x-=8)
              putchar((fse->fse_DosType >> x) & 0xFF);

         putchar((fse->fse_DosType & 0xFF) + 0x30);

         printf("\n                    Version: %d",(fse->fse_Version >> 16));
         printf(".%ld\n\n",(fse->fse_Version & 0xFFFF));
         }
     }
}
```

Additional programming information on the FileSystem resource can be found in the include files and the Autodocs for the FileSystem resource in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* and the "Expansion " chapter of the *Amiga ROM Kernel Reference Manual: Libraries*.

FileSystem Resource Information	
INCLUDES	resources/filesysres.h resources/filesysres.i
AUTODOCS	filesysres.doc
LIBRARIES	expansion library

Misc Resource

The misc resource oversees usage of the serial data port, the serial communication bits, the parallel data and handshake port, and the parallel communication bits. Before using serial or parallel port hardware, it first must be acquired from the misc resource.

The misc resource provides two functions for allocating and freeing the serial and parallel hardware.

Misc Resource Functions

AllocMiscResource()	Allocate one of the serial or parallel misc resources.
FreeMiscResource()	Deallocate one of the serial or parallel misc resources.

Once you've successfully allocated one of the misc resources, you are free to write directly to its hardware locations. Information on the serial and parallel hardware can be found in the *Amiga Hardware Reference Manual* and the *hardware/custom.h* include file.

The two examples below are assembly and C versions of the same code for locking the serial misc resources and waiting for CTRL-C to be pressed before releasing them.

ASSEMBLY EXAMPLE OF ALLOCATING MISC RESOURCES

```
* Alloc_Misc.a
*
* Assembly language fragment that grabs the two parts of the serial
* resource (using misc.resource).  If it gets the resource, it will
* wait for CTRL-C to be pressed before releasing.
*
* While we are waiting, the query_serial program should be run.  It will try
* to open the serial device and if unsuccessful, will return the name of the
* owner.  It will be us, Serial Port Hog!
*
* When a task has successfully obtained the serial resource, it "owns"
* the hardware registers that control the serial port.  No other tasks
* are allowed to interfere.
*
* Assemble with Adapt
*     HX68 Allocate_Misc.a to Allocate_Misc.o
*
* Link
*     Blink FROM Allocate_Misc.o TO Allocate_Misc LIB LIB:amiga.lib
*

                INCDIR  "include:"
                INCLUDE "exec/types.i"
                INCLUDE "resources/misc.i"
                INCLUDE "dos/dos.i"

        xref    _AbsExecBase     ; We get this from outside...
        xref    _LVOOpenResource ; We get this from outside...
        xref    _LVOWait         ; We get this from outside...

;
; Open Exec and the misc.resource, check for success
;
                move.l  _AbsExecBase,a6         ;Prepare to use exec
                lea.l   MiscName(pc),a1
                jsr     _LVOOpenResource(a6)    ;Open "misc.resource"
                move.l  d0,d7                   ;Stash resource base
                bne.s   resource_ok
                moveq   #RETURN_FAIL,d0
                rts

resource_ok     exg.l   d7,a6                   ;Put resource base in A6
```

```
;
; We now have a pointer to a resource.
; Call one of the resource's library-like vectors.
;
                move.l  #MR_SERIALBITS,d0       ;We want these bits
                lea.l   MyName(pc),a1           ;This is our name
                jsr     MR_ALLOCMISCRESOURCE(a6)
                tst.l   d0
                bne.s   no_bits                 ;Someone else has it...
                move.l  #MR_SERIALPORT,d0
                lea.l   MyName(pc),a1
                jsr     MR_ALLOCMISCRESOURCE(a6)
                tst.l   d0
                bne.s   no_port                 ;Someone else has it...
;
; We just stole the serial port registers; wait.
; Nobody else can use the serial port, including the serial.device!
;
                exg.l   d7,a6                   ;use exec again
                move.l  #SIGBREAKF_CTRL_C,d0
                jsr     _LVOWait(a6)            ;Wait for CTRL-C
                exg.l   d7,a6                   ;Get resource base back
;
; Free 'em up
;
                move.l  #MR_SERIALPORT,d0
                jsr     MR_FREEMISCRESOURCE(a6)
no_port
                move.l  #MR_SERIALBITS,d0
                jsr     MR_FREEMISCRESOURCE(a6)
no_bits
                moveq   #RETURN_FAIL,d0
                rts
;
; Text area
;
MiscName        dc.b    'misc.resource',0
MyName          dc.b    'Serial Port Hog',0
                dc.w    0
                END
```

C EXAMPLE OF ALLOCATING MISC RESOURCES

```
/*
 * Allocate_Misc.c
 *
 * Example of allocating a miscellaneous resource
 * We will allocate the serial resource and wait till
 * CTRL-C is pressed.  While we are waiting, the
 * query_serial program should be run.  It will try
 * to open the serial device and if unsuccessful, will
 * return the name of the owner.  It will be us!
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <resources/misc.h>

#include <clib/exec_protos.h>
#include <clib/misc_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif
```

```
struct Library *MiscBase = NULL;

void main(int argc, char **argv)
{
UBYTE *owner = NULL;        /* owner of misc resource */

if (!(MiscBase= (struct Library *)OpenResource(MISCNAME)))
    printf("Cannot open %s\n",MISCNAME);
else
   {
    /* Allocate both pieces of the serial hardware */
    if ((owner = AllocMiscResource(MR_SERIALPORT,"Serial Port Hog")) == NULL)
        {
        if ((owner = AllocMiscResource(MR_SERIALBITS,"Serial Port Hog")) == NULL)
            {
            /* Wait for CTRL-C to be pressed */
            printf("\nWaiting for CTRL-C...\n");
            Wait(SIGBREAKF_CTRL_C);

            /* We're back */

            /* Deallocate the serial port register */
            FreeMiscResource(MR_SERIALBITS);
            }
        else
            printf("\nUnable to allocate MR_SERIALBITS because %s owns it\n",owner);

        /* Deallocate the serial port */
        FreeMiscResource(MR_SERIALPORT);
       }
    else
       printf("\nUnable to allocate MR_SERIALPORT because %s owns it\n",owner);
   }
}
```

The example below will try to open the serial device and execute the SDCMD_QUERY command. If it cannot open the serial device, it will do an **AllocMiscResource()** on the serial port and return the name of the owner.

```
/*
 * Query_Serial.c
 *
 * We will try to open the serial device and if unsuccessful,
 * will return the name of the owner.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <resources/misc.h>
#include <devices/serial.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/dos_protos.h>
#include <clib/misc_protos.h>

#include <stdio.h>
#include <stdlib.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }      /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }   /* really */
#endif

struct Library *MiscBase;

struct MsgPort  *SerialMP;          /* Message port pointer */
struct IOExtSer *SerialIO;          /* I/O request pointer */
```

```
void main(void)
{
UWORD status;     /* return value of SDCMD_QUERY */
UBYTE *user;      /* name of serial port owner if not us */

if (SerialMP=CreatePort(NULL,NULL) )
    {
    if (SerialIO=(struct IOExtSer *)CreateExtIO(SerialMP,sizeof(struct IOExtSer)) )
        {
        if (OpenDevice(SERIALNAME,0L,(struct IORequest *)SerialIO,0))
            {
            printf("\n%s did not open",SERIALNAME);

            MiscBase= (struct Library *)OpenResource(MISCNAME);

            /* Find out who has the serial device */
            if ((user = AllocMiscResource(MR_SERIALPORT,"Us")) == NULL)
                {
                printf("\n");
                FreeMiscResource(MR_SERIALPORT);
                }
            else
                printf(" because %s owns it \n\n",user);
            }
        else
            {
            SerialIO->IOSer.io_Command  = SDCMD_QUERY;
            DoIO((struct IORequest *)SerialIO);          /* execute query */

            status = SerialIO->io_Status;                 /* store returned status */

            printf("\tThe serial port status is %x\n",status);

            CloseDevice((struct IORequest *)SerialIO);
            }

        DeleteExtIO(SerialIO);
        }
    else
        printf("Can't create I/O request\n");

    DeletePort(SerialMP);
    }
else
    printf("Can't create message port\n");
}
```

Take Over Everything. There are two serial.device resources to take over, MR_SERIALBITS and MR_SERIALPORT. You should get both resources when you take over the serial port to prevent other tasks from using them. The parallel.device also has two resources to take over. See the *resources/misc.h* include file for the relevant definitions and structures.

Under V1.3 and earlier versions of the Amiga system software the MR_GETMISCRESOURCE routine will always fail if the serial device has been used at all by another task (even if that task has finished using the resource. In other words, once a printer driver or communication package has been activated, it will keep the associated resource locked up preventing your task from using it. Under these conditions, you must get the resource back from the system yourself.

You do this by calling the function FlushDevice():

```
/*
 * A safe way to expunge ONLY a certain device.  The serial.device holds
 * on to the misc serial resource until a general expunge occurs.
 * This code attempts to flush ONLY the named device out of memory and
 * nothing else.  If it fails, no status is returned since it would have
 * no valid use after the Permit().
 */
```

```
#include <exec/types.h>
#include <exec/execbase.h>

#include <clib/exec_protos.h>

void FlushDevice(char *);

extern struct ExecBase *SysBase;

void FlushDevice(char *name)
{
struct Device *devpoint;

Forbid();

if (devpoint=(struct Device *)FindName(&SysBase->DeviceList,name) )
    RemDevice(devpoint);

Permit();
}
```

Additional programming information on the misc resource can be found in the include files and the Autodocs for the misc resource.

Misc Resource Information	
INCLUDES	resources/misc.h resources/misc.i hardware/custom.h hardware/custom.i
AUTODOCS	misc.doc

Potgo Resource

The potgo resource is used to get control of the hardware POTGO register connected to the proportional I/O pins on the game controller ports. There are two registers, POTGO (write-only) and POTINP (read-only). These pins could also be used for digital I/O.

The potgo resource provides three functions for working with the POTGO hardware.

	Potgo Resource Functions
AllocPotBits()	Allocate bits in the POTGO register.
FreePotBits()	Free previously allocated bits in the POTGO register.
WritePotgo()	Set and clear bits in the POTGO register. The bits must have been allocated before calling this function.

The example program shown below demonstrates how to use the ptogo resource to track mouse button presses on port 1.

```
/*
 * Read_Potinp.c
 *
 * An example of using the potgo.resource to read pins 9 and 5 of
 * port 1 (the non-mouse port).  This bypasses the gameport.device.
 * When the right or middle button on a mouse plugged into port 1 is pressed,
 * the read value will change.
 *
 * Use of port 0 (mouse) is unaffected.
 *
 * Compile with SAS C 5.10  lc -b1 -cfistq -v -y -L
 *
 * Run from CLI only
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <resources/potgo.h>
#include <hardware/custom.h>

#include <clib/exec_protos.h>
#include <clib/potgo_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) {return(0);}  /* Disable SAS Ctrl-C checking */
int chkabort(void) { return(0); }  /* really */
#endif

struct PotgoBase *PotgoBase;
ULONG potbits;
UWORD value;

#define UNLESS(x) if(!(x))
#define UNTIL(x)  while(!(x))

#define OUTRY 1L<<15
#define DATRY 1L<<14
#define OUTRX 1L<<13
#define DATRX 1L<<12

extern struct Custom far custom;

void main(int argc,char **argv)
{
UNLESS (PotgoBase=(struct PotgoBase *)OpenResource("potgo.resource"))
        return;

potbits=AllocPotBits(OUTRY|DATRY|OUTRX|DATRX);

/* Get the bits for the right and middle mouse buttons on the alternate mouse port. */

if (potbits != (OUTRY|DATRY|OUTRX|DATRX))
    {
    printf("Pot bits are already allocated! %lx\n",potbits);
    FreePotBits(potbits);
    return;
    }

/* Set all ones in the register (masked by potbits) */
WritePotgo(0xFFFFFFFFL,potbits);

printf("\nPlug a mouse into the second port.  This program will indicate when\n");
printf("the right or middle button (if the mouse is so equipped) is pressed.\n");
printf("Stop the program with Control-C. Press return now to begin.\n");

getchar();

UNTIL (SIGBREAKF_CTRL_C & SetSignal(0L,0L))
      /* until CTRL-C is pressed */
      {
      /* Read word at $DFF016 */
        value = custom.potinp;

      /* Show what was read (restricted to our allocated bits) */
      printf("POTINP = $%lx\n",value & potbits);
      }

FreePotBits(potbits);
}
```

Additional programming information on the potgo resource can be found in the include files and the Autodocs for the potgo resource.

Potgo Resource Information	
INCLUDES	resources/potgo.h resources/potgo.i utility/hooks.h utility/hooks.i
AUTODOCS	potgo.doc

appendix A
IFF: INTERCHANGE FILE FORMAT

One of the Amiga's strengths is the wide acceptance of several IFF specifications. Most notable is the ease with which graphic files (of form "ILBM") can be transferred among dozens of paint, animation and special effects packages. This ability to to easily share data between a variety of programs lets the user select the best program for a specific job rather than fighting the restritions of a single, all-in-one software package. Developers can market specialized applications that are good at a certain limited set of operations, and with the help of the multitasking Amiga operating system, create the effect of a large integrated system.

Any developer with a package that creates or reads data should use an existing IFF standard. If no current IFF form is suitable then the developer should contact other developers and users with similar needs and work out a new IFF form using the design principles specified in this appendix. To prevent conflicts, new IFF forms must be registered with Commodore before they are used. No additional restrictions are placed on the design of IFF forms aside from the general IFF syntax rules listed here.

Contents of the IFF Specification

A Quick Introduction to IFF

Jerry Morrison, Electronic Arts
10-17-88

IFF is the Amiga-standard "Interchange File Format", designed to work across many machines.

Why IFF?

Did you ever have this happen to your picture file?

You can't load it into another paint program.
You need a converter to adopt to "ZooPaint" release 2.0 or a new hardware feature.
You must "export" and "import" to use it in a page layout program.
You can't move it to another brand of computer.

What about interchanging musical scores, digitized audio, and other data? It seems the only thing that does interchange well is plain ASCII text files.

It's inexcusable. And yet this is "normal" in MS-DOS.

What is IFF?

IFF, the "Interchange File Format" standard, encourages multimedia interchange between different programs and different computers. It supports long-lived, extensible data. It's great for composite files like a page layout file that includes photos, an animation file that includes music, and a library of sound effects.

IFF is a 2-level standard. The first layer is the "wrapper" or "envelope" structure for all IFF files. Technically, it's the syntax. The second layer defines particular IFF file types such as ILBM (standard raster pictures), ANIM (animation), SMUS (simple musical score), and 8SVX (8-bit sampled audio voice).

IFF is also a design idea:

programs should use interchange formats for their everyday storage.

This way, users rarely need converters and import/export commands to change software releases, application programs, or hardware.

What's the trick?

File compatibility is easy to achieve if programmers let go of one notion—dumping internal data structures to disk. A program's internal data structures should really be suited to what the program does and how it works. What's "best" changes as the program evolves new functions and methods. But a disk format should be suited to storage and interchange.

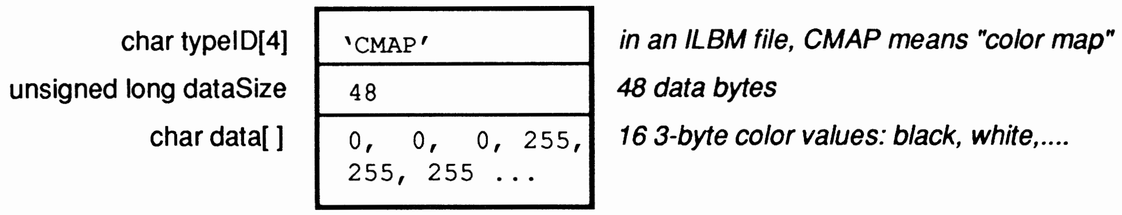
Once we design internal formats and disk formats for their own separate purposes, the rest is easy. Reading and writing become behind-the-scenes conversions. But two conversions hidden in each program is much better than a pile of conversion programs.

Does this seem strange? It's what ASCII text programs do! Text editors use line tables, piece tables, gaps, and other structures for fast editing and searching. Text generators and consumers construct and parse files. That's why the ASCII standard works so well.

Also, every file must be self-sufficient. E.g., a picture file has to include its size and number of bits/pixel.

What does an IFF file look like?

IFF is based on data blocks called "chunks". Here's an example color map chunk:



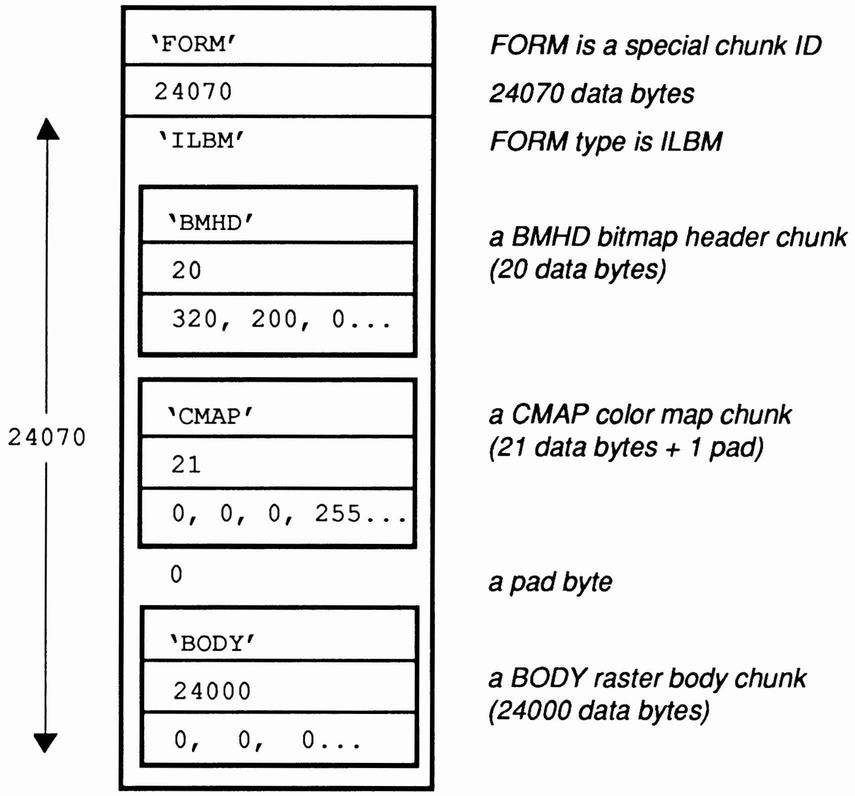
A chunk is made of a 4-character type identifier, a 32 bit data byte count, and the data bytes. It's like a Macintosh "resource" with a 32-bit size.

Fine points:

- Every 16- and 32-bit number is stored in 68000 byte order—highest byte first.
- An Intel CPU must reverse the 2- or 4-byte sequence of each number. This applies to chunk dataSize fields and to numbers inside chunk data. It does not affect character strings and byte data because you can't reverse a 1-byte sequence. But it does affect the 32-bit math used in IFF's MakeID macro. The standard does allow CPU specific byte ordering hidden within a chunk itself, but the practice is discouraged.
- Every 16- and 32-bit number is stored on an even address.
- Every odd-length chunk must be followed by a 0 pad byte. This pad byte is not counted in dataSize.
- An ID is made of 4 ASCII characters in the range " " (space, hex 20) through "~" (tilde, hex 7E). Leading spaces are not permitted.
- IDs are compared using a quick 32-bit equality test. Case matters.

A chunk typically holds a C structure, Pascal record, or an array. For example, an 'ILBM' picture has a 'BMHD' bitmap header chunk (a structure) and a 'BODY' raster body chunk (an array).

To construct an IFF file, just put a file type ID (like 'ILBM') into a wrapper chunk called a 'FORM' (Think "FILE"). Inside that wrapper place chunks one after another (with pad bytes as needed). The chunk size always tells you how many more bytes you need to skip over to get to the next chunk.


'FORM'
24070
'ILBM'
'BMHD'
20
320, 200, 0...
'CMAP'
21
0, 0, 0, 255...
0
'BODY'
24000
0, 0, 0...
24070
FORM is a special chunk ID
24070 data bytes
FORM type is ILBM
a BMHD bitmap header chunk (20 data bytes)
a CMAP color map chunk (21 data bytes + 1 pad)
a pad byte
a BODY raster body chunk (24000 data bytes)

A FORM always contains one 4-character FORM type ID (a file type, in this case 'ILBM') followed by any number of data chunks. In this example, the FORM type is 'ILBM', which stands for InterLeaved Bitmap. (ILBM is an IFF standard for bitplane raster pictures.) This example has 3 chunks. Note the pad byte after the odd length chunk.

Within FORMs ILBM, 'BMHD' identifies a bitmap header chunk, 'CMAP' a color map, and 'BODY' a raster body. In general, the chunk IDs in a FORM are local to the FORM type ID. The exceptions are the 4 global chunk IDs 'FORM', 'LIST', 'CAT ', and 'PROP'. (A FORM may contain other FORM chunks. E.g., an animation FORM might contain picture FORMs and sound FORMs.)

How to read an IFF file?

Example code and modules are provided for reading IFF files using iffparse.library. However, if you wish to read a non-complex FORM by hand, the following logic can be used.

Once you have entered the FORM (for example, the FORM ILBM shown above), stored the FORM length (24070 in the ILBM example) and are positioned on the first chunk, you may:

```
Loop: (until end-of-file or end-of-form)

          - Read the 4-character identifier of the chunk
          - Read the 32-bit (4 byte) chunklength
          - Decide if you want that chunk
              If yes, read chunklength bytes into destination structure
                      or buffer
              If no, seek forward chunklength bytes
          - If chunklength is odd, seek one more byte
```

Every IFF file is a 'FORM', 'LIST', or 'CAT ' chunk. You can recognize an IFF file by those first 4 bytes. ('FORM' is far and away the most common. We'll get to LIST and CAT below.) If the file contains a FORM, dispatch on the FORM type ID to a chunk-reader loop like the one above.

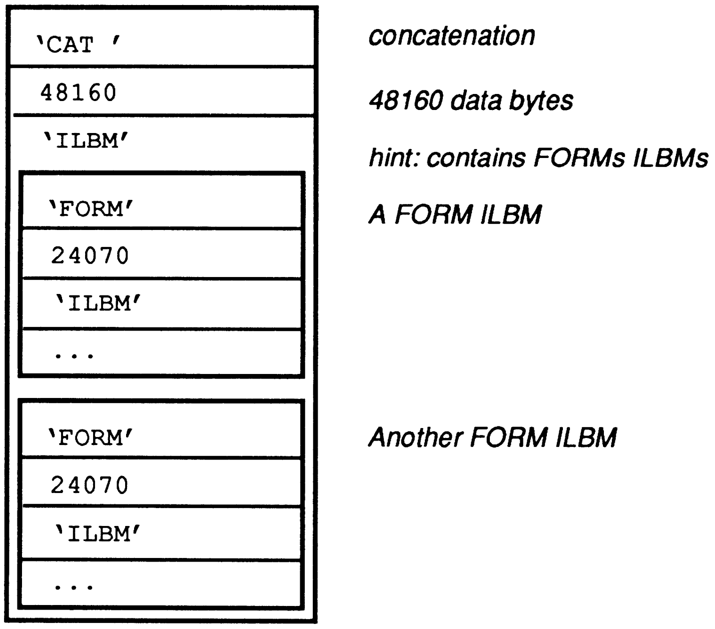
File extensibility

IFF files are extensible and forward/backward compatible:

- Chunk contents should be designed for compatibility across and for longevity. Every chunk should have a path for expansion; at minimum this will be an unused bit or two.
- The standards team for a FORM type can extend one of the chunks that contains a structure by appending new, optional structure fields.
- Anyone can define new FORM types as well as new chunk types within a FORM type. Storing private chunks within a FORM is OK, but be sure to register your activities with Commodore Applications and Technical Support.
- A chunk can be superseded by a new chunk type, e.g., to store more bits per RGB color register. New programs can output the old chunk (for backward compatibility) along with the new chunk.
- If you must change data in an incompatible way, change the chunk ID or the FORM type ID.

Advanced Topics: CAT, LIST, and PROP (not all that important)

Sometimes you want to put several "files" into one, such as a picture library. This is what CAT is for. It "concatenates" FORM and LIST chunks.



This example CAT holds two ILBMs. It can be shown outline-style:

```
CAT ILBM
..FORM ILBM   \
....BMHD      | a complete FORM ILBM picture
....CMAP      |
....BODY      /
..FORM ILBM
....BMHD
....CMAP
....BODY
```

Sometimes you want to share the same color map across many pictures. LIST and PROP do this:

```
LIST ILBM
..PROP ILBM   default properties for FORMs ILBM
....CMAP      an ILBM CMAP chunk (there could be a BMHD chunk here, too)
..FORM ILBM
....BMHD      (there could be a CMAP here to override the default)
....BODY
..FORM ILBM
....BMHD      (there could be a CMAP here to override the default)
....BODY
```

A LIST holds PROPs and FORMs (and occasionally LISTs and CATs). A PROP ILBM contains default data (in the above example, just one CMAP chunk) for all FORMs ILBM in the LIST. Any FORM may override the PROP-defined default with its own CMAP. All PROPs must appear at the beginning of a LIST. Each FORM type defines as standard (among other things) which of its chunks are "property chunks" (may appear in PROPs) and which are "data chunks" (may not appear in PROPs).

"EA IFF 85" Standard for Interchange Format Files

Document Date:	January 14, 1985 (Updated Oct, 1988 Commodore-Amiga, Inc.)
From:	Jerry Morrison, Electronic Arts
Status Of Standard:	Released to the public domain, and in use

1. Introduction

Standards are Good for Software Developers

As home computer hardware evolves into better and better media machines, the demand increases for higher quality, more detailed data. Data development gets more expensive, requires more expertise and better tools, and has to be shared across projects. Think about several ports of a product on one CD-ROM with 500M Bytes of common data!

Development tools need standard interchange file formats. Imagine scanning in images of "player" shapes, transferring them to an image enhancement package, moving them to a paint program for touch up, then incorporating them into a game. Or writing a theme song with a Macintosh score editor and incorporating it into an Amiga game. The data must at times be transformed, clipped, filled out, and moved across machine kinds. Media projects will depend on data transfer from graphic, music, sound effect, animation, and script tools.

Standards are Good for Software Users

Customers should be able to move their own data between independently developed software products. And they should be able to buy data libraries usable across many such products. The types of data objects to exchange are open-ended and include plain and formatted text, raster and structured graphics, fonts, music, sound effects, musical instrument descriptions, and animation.

The problem with expedient file formats—typically memory dumps is that they're too provincial. By designing data for one particular use (such as a screen snapshot), they preclude future expansion (would you like a full page picture? a multi-page document?). In neglecting the possibility that other programs might read their data, they fail to save contextual information (how many bit planes? what resolution?). Ignoring that other programs might create such files, they're intolerant of extra data (a different picture editor may want to save a texture palette with the image), missing data (such as no color map), or minor variations (perhaps a smaller image). In practice, a filed representation should rarely mirror an in-memory representation. The former should be designed for longevity; the latter to optimize the manipulations of a particular program. The same filed data will be read into different memory formats by different programs.

The IFF philosophy: "A little behind-the-scenes conversion when programs read and write files is far better than NxM explicit conversion utilities for highly specialized formats".

So we need some standardization for data interchange among development tools and products. The more developers that adopt a standard, the better for all of us and our customers.

Here is "EA IFF 1985"

Here is our offering: Electronic Arts' IFF standard for Interchange File Format. The full name is "EA IFF 1985". Alternatives and justifications are included for certain choices. Public domain subroutine packages and utility programs are available to make it easy to write and use IFF-compatible programs.

Part 1 introduces the standard. Part 2 presents its requirements and background. Parts 3, 4, and 5 define the primitive data types, FORMs, and LISTs, respectively, and how to define new high level types. Part 6 specifies the top level file structure. Section 7 lists names of the group responsible for this standard. Appendix A is included for quick reference and Appendix B.

References

American National Standard Additional Control Codes for Use with ASCII, ANSI standard 3.64-1979 for an 8-bit character set. See also ISO standard 2022 and ISO/DIS standard 6429.2.

The C Programming Language, Brian W. Kernighan and Dennis M. Ritchie, Bell Laboratories. Prentice-Hall, Englewood Cliffs, NJ, 1978.

C, A Reference Manual, Samuel P. Harbison and Guy L. Steele Jr., Tartan Laboratories. Prentice-Hall, Englewood Cliffs, NJ, 1984.

Compiler Construction, An Advanced Course, edited by F. L. Bauer and J. Eickel (Springer-Verlag, 1976). This book is one of many sources for information on recursive descent parsing.

DIF Technical Specification © 1981 by Software Arts, Inc. DIF™ is the format for spreadsheet data interchange developed by Software Arts, Inc. DIF™ is a trademark of Software Arts, Inc.

"FTXT" IFF Formatted Text, from Electronic Arts. IFF supplement document for a text format.

"ILBM" IFF Interleaved Bitmap, from Electronic Arts. IFF supplement document for a raster image format.

M68000 16/32-Bit Microprocessor Programmer's Reference Manual © 1984, 1982, 1980, 1979 by Motorola, Inc.

PostScript Language Manual © 1984 Adobe Systems Incorporated.
PostScript™ is a trademark of Adobe Systems, Inc.
Times and Helvetica® are registered trademarks of Allied Corporation.

Inside Macintosh © 1982, 1983, 1984, 1985 Apple Computer, Inc., a programmer's reference manual.
Apple® is a trademark of Apple Computer, Inc.
MacPaint™ is a trademark of Apple Computer, Inc.
Macintosh™ is a trademark licensed to Apple Computer, Inc.

InterScript: A Proposal for a Standard for the Interchange of Editable Documents © 1984 Xerox Corporation. Introduction to InterScript © 1985 Xerox Corporation.

2. Background for Designers

Part 2 is about the background, requirements, and goals for the standard. It's geared for people who want to design new types of IFF objects. People just interested in using the standard may wish to quickly scan this section.

What Do We Need?

A standard should be long on prescription and short on overhead. It should give lots of rules for designing programs and data files for synergy. But neither the programs nor the files should cost too much more than the expedient variety. Although we are looking to a future with CD-ROMs and perpendicular recording, the standard must work well on floppy disks.

For program portability, simplicity, and efficiency, formats should be designed with more than one implementation style in mind. It ought to be possible to read one of many objects in a file without scanning all the preceding data. (In practice, pure stream I/O is adequate although random access makes it easier to write files.) Some programs need to read and play out their data in real time, so we need good compromises between generality and efficiency.

As much as we need standards, they can't hold up product schedules. So we also need a kind of decentralized extensibility where any software developer can define and refine new object types without some "standards authority" in the loop. Developers must be able to extend existing formats in a forward- and backward-compatible way. A central repository for design information and example programs can help us take full advantage of the standard.

For convenience, data formats should heed the restrictions of various processors and environments. For example, word-alignment greatly helps 68000 access at insignificant cost to 8088 programs.

Other goals include the ability to share common elements over a list of objects and the ability to construct composite objects.

And finally, "Simple things should be simple and complex things should be possible" - Alan Kay.

Think Ahead

Let's think ahead and build programs that read and write files for each other and for programs yet to be designed. Build data formats to last for future computers so long as the overhead is acceptable. This extends the usefulness and life of today's programs and data.

To maximize interconnectivity, the standard file structure and the specific object formats must all be general and extensible. Think ahead when designing an object. File formats should serve many purposes and allow many programs to store and read back all the information they need; even squeeze in custom data. Then a programmer can store the available data and is encouraged to include fixed contextual details. Recipient programs can read the needed parts, skip unrecognized stuff, default missing data, and use the stored context to help transform the data as needed.

Scope

IFF addresses these needs by defining a standard file structure, some initial data object types, ways to define new types, and rules for accessing these files. We can accomplish a great deal by writing programs according to this standard, but do not expect direct compatibility with existing software. We'll need conversion programs to bridge the gap from the old world.

IFF is geared for computers that readily process information in 8-bit bytes. It assumes a "physical layer" of data storage and transmission that reliably maintains "files" as sequences of 8-bit bytes. The standard treats a "file" as a container of data bytes and is independent of how to find a file and whether it has a byte count.

This standard does not by itself implement a clipboard for cutting and pasting data between programs. A clipboard needs software to mediate access, and provide a notification mechanism so updates and requests for data can be detected.

Data Abstraction

The basic problem is *how to represent information* in a way that's program-independent, compiler-independent, machine-independent, and device-independent.

The computer science approach is "data abstraction", also known as "objects", "actors", and "abstract data types". A data abstraction has a "concrete representation" (its storage format), an "abstract representation" (its capabilities and uses), and access procedures that isolate all the calling software from the concrete representation. Only the access procedures touch the data storage. Hiding mutable details behind an interface is called "information hiding". What is hidden are the non-portable details of implementing the object, namely the selected storage representation and algorithms for manipulating it.

The power of this approach is modularity. By adjusting the access procedures we can extend and restructure the data without impacting the interface or its callers. Conversely, we can extend and restructure the interface and callers without making existing data obsolete. It's great for interchange!

But we seem to need the opposite: fixed file formats for all programs to access. Actually, we could file data abstractions ("filed objects") by storing the data and access procedures together. We'd have to encode the access procedures in a standard machine-independent programming language à la PostScript. Even with this, the interface can't evolve freely since we can't update all copies of the access procedures. So we'll have to design our abstract representations for limited evolution and occasional revolution (conversion).

In any case, today's microcomputers can't practically store true data abstractions. They can do the next best thing: store arbitrary types of data in "data chunks", each with a type identifier and a length count. The type identifier is a reference by name to the access procedures (any local implementation). The length count enables storage-level object operations like "copy" and "skip to next" independent of object type or contents.

Chunk writing is straightforward. Chunk reading requires a trivial parser to scan each chunk and dispatch to the proper access/conversion procedure. Reading chunks nested inside other chunks may require recursion, but no look ahead or backup.

That's the main idea of IFF. There are, of course, a few other details....

Previous Work

Where our needs are similar, we borrow from existing standards.

Our basic need to move data between independently developed programs is similar to that addressed by the Apple Macintosh desk scrap or "clipboard" [Inside Macintosh chapter "Scrap Manager"]. The Scrap Manager works closely with the Resource Manager, a handy filer and swapper for data objects (text strings, dialog window templates, pictures, fonts) including types yet to be designed [Inside Macintosh chapter "Resource Manager"]. The Resource Manager is akin to Smalltalk's object swapper.

We will probably write a Macintosh desk accessory that converts IFF files to and from the Macintosh clipboard for quick and easy interchange with programs like MacPaint and Resource Mover.

Macintosh uses a simple and elegant scheme of four-character "identifiers" to identify resource types, clipboard format types, file types, and file creator programs. Alternatives are unique ID numbers assigned by a central authority or by hierarchical authorities, unique ID numbers generated by algorithm, other fixed length character strings, and variable length strings. Character string identifiers double as readable signposts in data files and programs. The choice of 4 characters is a good tradeoff between storage space, fetch/compare/store time, and name space size. We'll honor Apple's designers by adopting this scheme.

"PICT" is a good example of a standard structured graphics format (including raster images) and its many uses [Inside Macintosh chapter "QuickDraw"]. Macintosh provides QuickDraw routines in ROM to create, manipulate, and display PICTs. Any application can create a PICT by simply asking QuickDraw to record a sequence of drawing commands. Since it's just as easy to ask QuickDraw to render a PICT to a screen or a printer, it's very effective to pass them between programs, say from an illustrator to a word processor. An important feature is the ability to store "comments" in a PICT which QuickDraw will ignore. (Actually, it passes them to your optional custom "comment handler".)

PostScript, Adobe System's print file standard, is a more general way to represent any print image (which is a specification for putting marks on paper) [PostScript Language Manual]. In fact, PostScript is a full-fledged programming language. To interpret a PostScript program is to render a document on a raster output device. The language is defined in layers: a lexical layer of identifiers, constants, and operators; a layer of reverse polish semantics including scope rules and a way to define new subroutines; and a printing-specific layer of built-in identifiers and operators for rendering graphic images. It is clearly a powerful (Turing equivalent) image definition language. PICT and a subset of PostScript are candidates for structured graphics standards.

A PostScript document can be printed on any raster output device (including a display) but cannot generally be edited. That's because the original flexibility and constraints have been discarded. Besides, a PostScript program may use arbitrary computation to supply parameters like placement and size to each operator. A QuickDraw PICT, in comparison, is a more restricted format of graphic primitives parameterized by constants. So a PICT can be edited at the level of the primitives, e.g., move or thicken a line. It cannot be edited at the higher level of, say, the bar chart data which generated the picture.

PostScript has another limitation: not all kinds of data amount to marks on paper. A musical instrument description is one example. PostScript is just not geared for such uses.

"DIF" is another example of data being stored in a general format usable by future programs [DIF Technical Specification]. DIF is a format for spreadsheet data interchange. DIF and PostScript are both expressed in plain ASCII text files. This is very handy for printing, debugging, experimenting, and transmitting across modems. It can have substantial cost in compaction and read/write work, depending on use. We won't store IFF files this way but we could define an ASCII alternate representation with a converter program.

InterScript is the Xerox standard for interchange of editable documents [Introduction to InterScript]. It approaches a harder problem: How to represent editable word processor documents that may contain formatted text, pictures, cross-references like figure numbers, and even highly specialized objects like mathematical equations? InterScript aims to define one standard representation for each kind of information. Each InterScript-compatible editor is supposed to preserve the objects it doesn't understand and even maintain nested cross-references. So a simple word processor would let you edit the text of a fancy document without discarding the equations or disrupting the equation numbers.

Our task is similarly to store high level information and preserve as much content as practical while moving it between programs. But we need to span a larger universe of data types and cannot expect to centrally define them all. Fortunately, we don't need to make programs preserve information that they don't understand. And for better or worse, we don't have to tackle general-purpose cross-references yet.

3. Primitive Data Types

Atomic components such as integers and characters that are interpretable directly by the CPU are specified in one format for all processors. We chose a format that's the same as used by the Motorola MC68000 processor [M68000 16/32-Bit Microprocessor Programmer's Reference Manual]. The high byte and high word of a number are stored *first.*

N.B.: Part 3 dictates the format for "primitive" data types where—and only where—used in the overall file structure. The number of such occurrences of dictated formats will be small enough that the costs of conversion, storage, and management of processor-specific files would far exceed the costs of conversion during I/O by "foreign" programs. A particular data chunk may be specified with a different format for its internal primitive types or with processor or environment specific variants if necessary to optimize local usage. Since that hurts data interchange, it's not recommended. (Cf. Designing New Data Sections, in Part 4.)

Alignment

All data objects larger than a byte are aligned on even byte addresses relative to the start of the file. This may require padding. Pad bytes are to be written as zeros, but don't count on that when reading.

This means that every odd-length "chunk" must be padded so that the next one will fall on an even boundary. Also, designers of structures to be stored in chunks should include pad fields where needed to align every field larger than a byte. For best efficiency, long word data should be arranged on long word (4 byte) boundaries. Zeros should be stored in all the pad bytes.

Justification: Even-alignment causes a little extra work for files that are used only on certain processors but allows 68000 programs to construct and scan the data in memory and do block I/O. Any 16-bit or greater CPU will have faster access to aligned data. You just add an occasional pad field to data structures that you're going to block read/write or else stream read/write an extra byte. And the same source code works on all processors. Unspecified alignment, on the other hand, would force 68000 programs to (dis)assemble word and long word data one byte at a time. Pretty cumbersome in a high level language. And if you don't conditionally compile that step out for other processors, you won't gain anything.

Numbers

Numeric types supported are two's complement binary integers in the format used by the MC68000 processor—high byte first, high word first—the reverse of 8088 and 6502 format.

```
UBYTE    8 bits unsigned
WORD    16 bits signed
UWORD   16 bits unsigned
LONG    32 bits signed
```

The actual type definitions depend on the CPU and the compiler. In this document, we'll express data type definitions in the C programming language. [See C, A Reference Manual.] In 68000 Lattice C:

```
typedef unsigned char   UBYTE;  /*  8 bits unsigned     */
typedef short           WORD;   /* 16 bits signed       */
typedef unsigned short  UWORD;  /* 16 bits unsigned     */
typedef long            LONG;   /* 32 bits signed       */
```

Characters

The following character set is assumed wherever characters are used, e.g., in text strings, IDs, and TEXT chunks (see below). Characters are encoded in 8-bit ASCII. Characters in the range NUL (hex 0) through DEL (hex 7F) are well defined by the 7-bit ASCII standard. IFF uses the graphic group " " (SP, hex 20) through "~" (hex 7E).

Most of the control character group hex 01 through hex 1F have no standard meaning in IFF. The control character LF (hex 0A) is defined as a "newline" character. It denotes an intentional line break, that is, a paragraph or line terminator. (There is no way to store an automatic line break. That is strictly a function of the margins in the environment the text is placed.) The control character ESC (hex 1B) is a reserved escape character under the rules of ANSI standard 3.64-1979 American National Standard Additional Control Codes for Use with ASCII, ISO standard 2022, and ISO/DIS standard 6429.2.

Characters in the range hex 7F through hex FF are not globally defined in IFF. They are best left reserved for future standardization. (Note that the FORM type FTXT (formatted text) defines the meaning of these characters within FTXT forms.) In particular, character values hex 7F through hex 9F are control codes while characters hex A0 through hex FF are extended graphic characters like ©, as per the ISO and ANSI standards cited above. [See the supplementary document "FTXT" IFF Formatted Text.]

Dates

A "creation date" is defined as the date and time a stream of data bytes was created. (Some systems call this a "last modified date".) Editing some data changes its creation date. Moving the data between volumes or machines does not.

The IFF standard date format will be one of those used in MS-DOS, Macintosh, or AmigaDOS (probably a 32-bit unsigned number of seconds since a reference point). Issue: Investigate these three.

Type IDs

A "type ID", "property name", "FORM type", or any other IFF identifier is a 32-bit value: the concatenation of four ASCII characters in the range " " (SP, hex 20) through "~" (hex 7E). Spaces (hex 20) should not precede printing characters; trailing spaces are OK. Control characters are forbidden.

```
        typedef CHAR ID[4];
```

IDs are compared using a simple 32-bit case-dependent equality test. FORM type IDs are restricted. Since they may be stored in filename extensions lower case letters and punctuation marks are forbidden. Trailing spaces are OK.

Carefully choose those four characters when you pick a new ID. Make them mnemonic so programmers can look at an interchange format file and figure out what kind of data it contains. The name space makes it possible for developers scattered around the globe to generate ID values with minimal collisions so long as they choose specific names like "MUS4" instead of general ones like "TYPE" and "FILE".

Commodore Applications and Technical Support has undertaken the task of maintaining the registry of FORM type IDs and format descriptions. See the IFF registry document for more information.

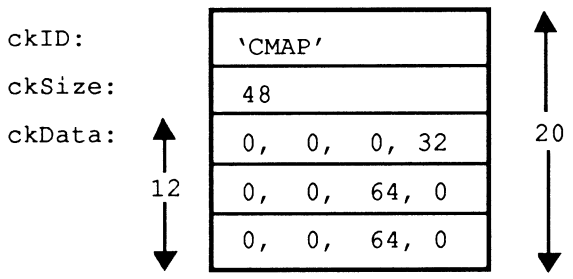
Sometimes it's necessary to make data format changes that aren't backward compatible. As much as we work for compatibility, unintended interactions can develop. Since IDs are used to denote data formats in IFF, new IDs are chosen to denote revised formats. Since programs won't read chunks whose IDs they don't recognize (see Chunks, below), the new IDs keep old programs from stumbling over new data. The conventional way to chose a "revision" ID is to increment the last character if it's a digit or else change the last character to a digit. E.g., first and second revisions of the ID "XY" would be "XY1" and "XY2". Revisions of "CMAP" would be "CMA1" and "CMA2".

Chunks

Chunks are the building blocks in the IFF structure. The form expressed as a C typedef is:

```
typedef struct {
    ID      ckID;                   /* 4 character ID */
    LONG    ckSize;                 /* sizeof(ckData) */
    UBYTE   ckData[/* ckSize */];
    } Chunk;
```

We can diagram an example chunk — a "CMAP" chunk containing 12 data bytes — like this:



That's 4 bytes of `ckID`, 4 bytes of `ckSize` and 12 data bytes. The total space used is 20 bytes.

The `ckID` identifies the format and purpose of the chunk. As a rule, a program must recognize `ckID` to interpret `ckData`. It should skip over all unrecognized chunks. The `ckID` also serves as a format version number as long as we pick new IDs to identify new formats of `ckData` (see above).

The following `ckIDs` are universally reserved to identify chunks with particular IFF meanings: "LIST", "FORM", "PROP", "CAT ", and "   ". The special ID "   " (4 spaces) is a `ckID` for "filler" chunks, that is, chunks that fill space but have no meaningful contents. The IDs "LIS1" through "LIS9", "FOR1" through "FOR9", and "CAT1" through "CAT9" are reserved for future "version number" variations. All IFF-compatible software must account for these chunk IDs.

The `ckSize` is a logical block size—how many data bytes are in `ckData`. If `ckData` is an odd number of bytes long, a 0 pad byte follows which is not included in `ckSize`. (Cf. Alignment.) A chunk's total physical size is `ckSize` rounded up to an even number plus the size of the header. So the smallest chunk is 8 bytes long with `ckSize` = 0. For the sake of following chunks, programs must respect every chunk's `ckSize` as a virtual end-of-file for reading its `ckData` even if that data is malformed, e.g., if nested contents are truncated.

We can describe the syntax of a chunk as a regular expression with "#" representing the `ckSize`, the length of the following braced bytes. The "[0]" represents a sometimes needed pad byte. (The regular expressions in this document are collected in Appendix A along with an explanation of notation.)

```
        Chunk    ::= ID #{ UBYTE* } [0]
```

One chunk output technique is to stream write a chunk header, stream write the chunk contents, then random access back to the header to fill in the size. Another technique is to make a preliminary pass over the data to compute the size, then write it out all at once.

Strings, String Chunks, and String Properties

In a string of ASCII text, linefeed (0x0A) denotes a forced line break (paragraph or line terminator). Other control characters are not used. (Cf. Characters.) For maximum compatibility with line editors, two linefeed characters are often used to indicate a paragraph boundary.

The `ckID` for a chunk that contains a string of plain, unformatted text is "TEXT". As a practical matter, a text string should probably not be longer than 32767 bytes. The standard allows up to 2^{31} - 1 bytes. The `ckID` "TEXT" is globally reserved for this use.

When used as a data property (see below), a text string chunk may be 0 to 255 characters long. Such a string is readily converted to a C string or a Pascal `STRING[255]`. The `ckID` of a property must have a unique property name, *not* "TEXT".

When used as a part of a chunk or data property, restricted C string format is normally used. That means 0 to 255 characters followed by a NULL byte (ASCII value 0).

Data Properties (advanced topic)

Data properties specify attributes for following (non-property) chunks. A data property essentially says "identifier = value", for example "XY = (10, 200)", telling something about following chunks. Properties may only appear inside data sections ("FORM" chunks, cf. Data Sections) and property sections ("PROP" chunks, cf. Group PROP).

The form of a data property is a type of Chunk. The `ckID` is a property name as well as a property type. The `ckSize` should be small since data properties are intended to be accumulated in RAM when reading a file. (256 bytes is a reasonable upper bound.) Syntactically:

```
        Property  ::=  Chunk
```

When designing a data object, use properties to describe context information like the size of an image, even if they don't vary in your program. Other programs will need this information.

Think of property settings as assignments to variables in a programming language. Multiple assignments are redundant and local assignments temporarily override global assignments. The order of assignments doesn't matter as long as they precede the affected chunks. (Cf. LISTs, CATs, and Shared Properties.)

Each object type (FORM type) is a local name space for property IDs. Think of a "CMAP" property in a "FORM ILBM" as the qualified ID "ILBM.CMAP". A "CMAP" inside some other type of FORM may not have the same meaning. Property IDs specified when an object type is designed (and therefore known to all clients) are called "standard" while specialized ones added later are "nonstandard".

Links

Issue: A standard mechanism for "links" or "cross references" is very desirable for things like combining images and sounds into animations. Perhaps we'll define "link" chunks within FORMs that refer to other FORMs or to specific chunks within the same and other FORMs. This needs further work. EA IFF 1985 has no standard link mechanism.

For now, it may suffice to read a list of, say, musical instruments, and then just refer to them within a musical score by sequence number.

File References

Issue: We may need a standard form for references to other files. A "file ref" could name a directory and a file in the same type of operating system as the reference's originator. Following the reference would expect the file to be on some mounted volume, or perhaps the same directory as the file that made the reference. In a network environment, a file reference could name a server, too.

Issue: How can we express operating-system independent file references?

Issue: What about a means to reference a portion of another file? Would this be a "file ref" plus a reference to a "link" within the target file?

4. Data Sections

The first thing we need of a file is to check: Does it contain IFF data and, if so, does it contain the kind of data we're looking for? So we come to the notion of a "data section".

A "data section" or IFF "FORM" is one self-contained "data object" that might be stored in a file by itself. It is one high level data object such as a picture or a sound effect, and generally contains a grouping of chunks. The IFF structure "FORM" makes it self-identifying. It could be a composite object like a musical score with nested musical instrument descriptions.

Group FORM

A data section is a chunk with `ckID` "FORM" and this arrangement:

```
FORM       ::= "FORM" #{ FormType (LocalChunk | FORM | LIST | CAT)* }
FormType   ::= ID
LocalChunk ::= Property | Chunk
```

The ID "FORM" is a syntactic keyword like "struct" in C. Think of a "struct ILBM" containing a field "CMAP". If you see "FORM" you will know to expect a FORM type ID (the structure name, "ILBM" in this example) and a particular contents arrangement or "syntax" (local chunks, FORMs, LISTs, and CATs). A "FORM ILBM", in particular, might contain a local chunk "CMAP", an "ILBM.CMAP" (to use a qualified name).

So the chunk ID "FORM" indicates a data section. It implies that the chunk contains an ID and some number of nested chunks. In reading a FORM, like any other chunk, programs must respect its `ckSize` as a virtual end-of-file for reading its contents, even if they're truncated.

The FORM type is a restricted ID that may not contain lower case letters or punctuation characters. (Cf. Type IDs. Cf. Single Purpose Files.)

The type-specific information in a FORM is composed of its "local chunks": data properties and other chunks. Each FORM type is a local name space for local chunk IDs. So "CMAP" local chunks in other FORM types may be unrelated to "ILBM.CMAP". More than that, each FORM type defines semantic scope. If you know what a FORM ILBM is, you will know what an ILBM.CMAP is.

Local chunks defined when the FORM type is designed (and therefore known to all clients of this type) are called "standard" while specialized ones added later are "nonstandard".

Among the local chunks, property chunks give settings for various details like text font while the other chunks supply the essential information. This distinction is not clear cut. A property setting can be cancelled by a later setting of the same property. E.g., in the sequence:

```
prop1 = x  (Data A)  prop1 = z  prop1 = y (Data B)
```

prop1 is = x for Data A, and y for Data B. The setting `prop1 = z` has no effect.

For clarity, the universally reserved chunk IDs "LIST", "FORM", "PROP", "CAT ", "   ", "LIS1" through "LIS9", "FOR1" through "FOR9", and "CAT1" through "CAT9" may not be FORM type IDs.

Part 5, below, talks about grouping FORMs into LISTs and CATs. They let you group a bunch of FORMs but don't impose any particular meaning or constraints on the grouping. Read on.

Composite FORMs

A FORM chunk inside a FORM is a full-fledged data section. This means you can build a composite object such as a multi-frame animation sequence by nesting available picture FORMs and sound effect FORMs. You can insert additional chunks with information like frame rate and frame count.

Using composite FORMs, you leverage on existing programs that create and edit the component FORMs. Those editors may even look into your composite object to copy out its type of component. Such editors are not allowed to replace their component objects within your composite object. That's because the IFF standard lets you specify consistency requirements for the composite FORM such as maintaining a count or a directory of the components. Only programs that are written to uphold the rules of your FORM type may create or modify such FORMs.

Therefore, in designing a program that creates composite objects, you are strongly requested to provide a facility for your users to import and export the nested FORMs. Import and export could move the data through a clipboard or a file.

Here are several existing FORM types and rules for defining new ones:

FTXT

An FTXT data section contains text with character formatting information like fonts and faces. It has no paragraph or document formatting information like margins and page headers. FORM FTXT is well matched to the text representation in Amiga's Intuition environment. See the supplemental document "FTXT" IFF Formatted Text.

ILBM

"ILBM" is an InterLeaved BitMap image with color map; a machine-independent format for raster images. FORM ILBM is the standard image file format for the Commodore-Amiga computer and is useful in other environments, too. See the supplemental document "ILBM" IFF Interleaved Bitmap.

PICS

The data chunk inside a "PICS" data section has ID "PICT" and holds a QuickDraw picture. Issue: Allow more than one PICT in a PICS? See Inside Macintosh chapter "QuickDraw" for details on PICTs and how to create and display them on the Macintosh computer.

The only standard property for PICS is "XY", an optional property that indicates the position of the PICT relative to "the big picture". The contents of an XY is a QuickDraw Point.

Note: PICT may be limited to Macintosh use, in which case there'll be another format for structured graphics in other environments.

Other Macintosh Resource Types

Some other Macintosh resource types could be adopted for use within IFF files; perhaps MWRT, ICN, ICN#, and STR#.

Issue: Consider the candidates and reserve some more IDs.

Designing New Data Sections

Supplemental documents will define additional object types. A supplement needs to specify the object's purpose, its FORM type ID, the IDs and formats of standard local chunks, and rules for generating and interpreting the data. It's a good idea to supply typedefs and an example source program that accesses the new object. See "ILBM" IFF Interleaved Bitmap for such an example.

Anyone can pick a new FORM type ID but should reserve it with Commodore Applications and Technical Support (CATS) at their earliest convenience. While decentralized format definitions and extensions are possible in IFF, our preference is to get design consensus by committee, implement a program to read and write it, perhaps tune the format before it becomes locked in stone, and then publish the format with example code. Some organization should remain in charge of answering questions and coordinating extensions to the format.

If it becomes necessary to incompatibly revise the design of some data section, its FORM type ID will serve as a version number (Cf. Type IDs). E.g., a revised "VDEO" data section could be called "VDE1". But try to get by with compatible revisions within the existing FORM type.

In a new FORM type, the rules for primitive data types and word-alignment (Cf. Primitive Data Types) may be overridden for the contents of its local chunks — but not for the chunk structure itself — if your documentation spells out the deviations. If machine-specific type variants are needed, e.g., to store vast numbers of integers in reverse bit order, then outline the conversion algorithm and indicate the variant inside each file, perhaps via different FORM types. Needless to say, variations should be minimized.

In designing a FORM type, encapsulate all the data that other programs will need to interpret your files. E.g., a raster graphics image should specify the image size even if your program always uses 320 x 200 pixels x 3 bitplanes. Receiving programs are then empowered to append or clip the image rectangle, to add or drop bitplanes, etc. This enables a lot more compatibility.

Separate the central data (like musical notes) from more specialized information (like note beams) so simpler programs can extract the central parts during read-in. Leave room for expansion so other programs can squeeze in new kinds of information (like lyrics). And remember to keep the property chunks manageably short—let's say $\leq$ 256 bytes.

When designing a data object, try to strike a good tradeoff between a super- general format and a highly-specialized one. Fit the details to at least one particular need, for example a raster image might as well store pixels in the current machine's scan order. But add the kind of generality that makes the format usable with foreseeable hardware and software. E.g., use a whole byte for each red, green, and blue color value even if this year's computer has only 4-bit video DACs. Think ahead and help other programs so long as the overhead is acceptable. E.g., run compress a raster by scan line rather than as a unit so future programs can swap images by scan line to and from secondary storage.

Try to design a general purpose "least common multiple" format that encompasses the needs of many programs without getting too complicated. Be sure to leave provisions for future expansion. Let's coalesce our uses around a few such formats widely separated in the vast design space. Two factors make this flexibility and simplicity practical. First, file storage space is getting very plentiful, so compaction is not always a priority. Second, nearly any locally-performed data conversion work during file reading and writing will be cheap compared to the I/O time.

It must be permitted to copy a LIST or FORM or CAT intact, e.g., to incorporate it into a composite FORM. So any kind of internal references within a FORM must be relative references. They could be relative to the start of the containing FORM, relative from the referencing chunk, or a sequence number into a collection.

With composite FORMs, you leverage on existing programs that create and edit the components. If you write a program that creates composite objects, please provide a facility for users to import and export the nested FORMs.

Finally, don't forget to specify all implied rules in detail.

5. LISTs, CATs, and Shared Properties (Advanced topics)

Data often needs to be grouped together, for example, consider a list of icons. Sometimes a trick like arranging little images into a big raster works, but generally they'll need to be structured as a first class group. The objects "LIST" and "CAT " are IFF-universal mechanisms for this purpose. Note: LIST and CAT are advanced topics the first time reader will want to skip.

Property settings sometimes need to be shared over a list of similar objects. E.g., a list of icons may share one color map. LIST provides a means called "PROP" to do this. One purpose of a LIST is to define the scope of a PROP. A "CAT ", on the other hand, is simply a concatenation of objects.

Simpler programs may skip LISTs and PROPs altogether and just handle FORMs and CATs. All "fully-conforming" IFF programs also know about "CAT ", "LIST", and "PROP". Any program that reads a FORM inside a LIST must process shared PROPs to correctly interpret that FORM.

Group CAT

A CAT is just an untyped group of data objects.

Structurally, a CAT is a chunk with chunk ID "CAT " containing a "contents type" ID followed by the nested objects. The `ckSize` of each contained chunk is essentially a relative pointer to the next one.

```
CAT             ::= "CAT " #{ ContentsType (FORM | LIST | CAT)* }
ContentsType    ::= ID  -- a hint or an "abstract data type" ID
```

In reading a CAT, like any other chunk, programs must respect its `ckSize` as a virtual end-of-file for reading the nested objects even if they're malformed or truncated.

The "contents type" following the CAT's `ckSize` indicates what kind of FORMs are inside. So a CAT of ILBMs would store "ILBM" there. It's just a hint. It may be used to store an "abstract data type". A CAT could just have blank contents ID ("   ") if it contains more than one kind of FORM.

CAT defines only the format of the group. The group's meaning is open to interpretation. This is like a list in LISP: the structure of cells is predefined but the meaning of the contents as, say, an association list depends on use. If you need a group with an enforced meaning (an "abstract datatype" or Smalltalk "subclass"), some consistency constraints, or additional data chunks, use a composite FORM instead (Cf. Composite FORMs).

Since a CAT just means a concatenation of objects, CATs are rarely nested. Programs should really merge CATs rather than nest them.

Group LIST

A LIST defines a group very much like CAT but it also gives a scope for PROPs (see below). And unlike CATs, LISTs should not be merged without understanding their contents.

Structurally, a LIST is a chunk with `ckID` "LIST" containing a "contents type" ID, optional shared properties, and the nested contents (FORMs, LISTs, and CATs), in that order. The `ckSize` of each contained chunk is a relative pointer to the next one. A LIST is not an arbitrary linked list—the cells are simply concatenated.

```
LIST         ::= "LIST" #{ ContentsType PROP* (FORM | LIST | CAT)* }
ContentsType ::= ID
```

Group PROP

PROP chunks may appear in LISTs (not in FORMs or CATs). They supply shared properties for the FORMs in that LIST. This ability to elevate some property settings to shared status for a list of forms is useful for both indirection and compaction. E.g., a list of images with the same size and colors can share one "size" property and one "color map" property. Individual FORMs can override the shared settings.

The contents of a PROP is like a FORM with no data chunks:

```
PROP    ::= "PROP" #{ FormType Property* }
```

It means, "Here are the shared properties for FORM type <FormType>".

A LIST may have at most one PROP of a FORM type, and all the PROPs must appear before any of the FORMs or nested LISTs and CATs. You can have subsequences of FORMs sharing properties by making each subsequence a LIST.

Scoping: Think of property settings as variable bindings in nested blocks of a programming language. In C this would look like:

```
#define Roman           0
#define Helvetica       1

void main()
        {
        int font=Roman; /* The global default */
                {
                printf("The font number is %d\n",font);
                }
                {
                int font=Helvetica;     /* local setting */
                printf("The font number is %d\n",font);
                }
                {
                printf("The font number is %d\n",font);
                }
        }


/*
 * Sample output:       The font number is 0
 *                      The font number is 1
 *                      The font number is 0
 */
```

An IFF file could contain:

```
LIST {
        PROP TEXT {
                FONT {TimesRoman}       /* shared setting       */
                }

        FORM TEXT {
                FONT {Helvetica}        /* local setting        */
                CHRS {Hello }           /* uses font Helvetica  */
                }

        FORM TEXT {
                CHRS {there.}           /* uses font TimesRoman */
                }
        }
```

The shared property assignments selectively override the reader's global defaults, but only for FORMs within the group. A FORM's own property assignments selectively override the global and group-supplied values. So when reading an IFF file, keep property settings on a stack. They are designed to be small enough to hold in main memory.

Shared properties are semantically equivalent to copying those properties into each of the nested FORMs right after their FORM type IDs.

Properties for LIST

Optional "properties for LIST" store the origin of the list's contents in a PROP chunk for the pseudo FORM type "LIST". They are the properties originating program "OPGM", processor family "OCPU", computer type "OCMP", computer serial number or network address "OSN ", and user name "UNAM". In our imperfect world, these could be called upon to distinguish between unintended variations of a data format or to work around bugs in particular originating/receiving program pairs. Issue: Specify the format of these properties.

A creation date could also be stored in a property, but let's ask that file creating, editing, and transporting programs maintain the correct date in the local file system. Programs that move files between machine types are expected to copy across the creation dates.

6. Standard File Structure

File Structure Overview

An IFF file is just a single chunk of type FORM, LIST, or CAT. Therefore an IFF file can be recognized by its first 4 bytes: "FORM", "LIST", or "CAT ". Any file contents after the chunk's end are to be ignored. (Some file transfer programs add garbage to the end of transferred files. This specification protects against such common damage).

The simplest IFF file would be one that does no more than encapsulate some binary data (perhaps even an old-fashioned single-purpose binary file). Here is a binary dump of such a minimal IFF example:

```
0000: 464F524D 0000001A 534E4150 43524143    FORM....SNAPCRAC
0010: 0000000D 68656C6C 6F2C776F 726C6421    ....hello,world!
0020: 0A00                                   ..
```

The first 4 bytes indicate this is a "FORM"; the most common IFF top level structure. The following 4 bytes indicate that the contents totals 26 bytes. The form type is listed as "SNAP".

Our form "SNAP" contains only one chunk at the moment; a chunk of type "CRAC". From the size ($0000000D) the amount of data must be 13 bytes. In this case, the data happens to correspond to the ASCII string "`hello, world!`<lf>". Since the number 13 is odd, a zero pad byte is added to the file. At any time new chunks could be added to form SNAP without affecting any other aspect of the file (other than the form size). It's that simple.

Since an IFF file can be a group of objects, programs that read/write single objects can communicate to an extent with programs that read/write groups. You're encouraged to write programs that handle all the objects in a LIST or CAT. A graphics editor, for example, could process a list of pictures as a multiple page document, one page at a time.

Programs should enforce IFF's syntactic rules when reading and writing files. Users should be told when a file is corrupt. This ensures robust data transfer. For minor damage, you may wish to give the user the option of using the suspect data, or cancelling. Presumably a user could read in a damaged file, then save whatever was salvaged to a valid file. The public domain IFF reader/writer subroutine package does some syntactic checks for you. A utility program"IFFCheck" is available that scans an IFF file and checks it for conformance to IFF's syntactic rules. IFFCheck also prints an outline of the chunks in the file, showing the `ckID` and `ckSize` of each. This is quite handy when building IFF programs. Example programs are also available to show details of reading and writing IFF files.

A merge program "IFFJoin" will be available that logically appends IFF files into a single CAT group. It "unwraps" each input file that is a CAT so that the combined file isn't nested CATs.

If we need to revise the IFF standard, the three anchoring IDs will be used as "version numbers". That's why IDs "FOR1" through "FOR9", "LIS1" through "LIS9", and "CAT1" through "CAT9" are reserved.

IFF formats are designed for reasonable performance with floppy disks. We achieve considerable simplicity in the formats and programs by relying on the host file system rather than defining universal grouping structures like directories for LIST contents. On huge storage systems, IFF files could be leaf nodes in a file structure like a B-tree. Let's hope the host file system implements that for us!

There are two kinds of IFF files: single purpose files and scrap files. They differ in the interpretation of multiple data objects and in the file's external type.

Single Purpose Files

A single purpose IFF file is for normal "document" and "archive" storage. This is in contrast with "scrap files" (see below) and temporary backing storage (non-interchange files).

The external file type (or filename extension, depending on the host file system) indicates the file's contents. It's generally the FORM type of the data contained, hence the restrictions on FORM type IDs.

Programmers and users may pick an "intended use" type as the filename extension to make it easy to filter for the relevant files in a filename requester. This is actually a "subclass" or "subtype" that conveniently separates files of the same FORM type that have different uses. Programs cannot demand conformity to its expected subtypes without overly restricting data interchange since they cannot know about the subtypes to be used by future programs that users will want to exchange data with.

Issue: How to generate 3-letter MS-DOS extensions from 4-letter FORM type IDs?

Most single purpose files will be a single FORM (perhaps a composite FORM like a musical score containing nested FORMs like musical instrument descriptions). If it's a LIST or a CAT, programs should skip over unrecognized objects to read the recognized ones or the first recognized one. Then a program that can read a single purpose file can read something out of a "scrap file", too.

Scrap Files (not currently used)

A "scrap file" is for maximum interconnectivity in getting data between programs; the core of a clipboard function. Scrap files may have type "IFF " or filename extension ".IFF".

A scrap file is typically a CAT containing alternate representations of the same basic information. Include as many alternatives as you can readily generate. This redundancy improves interconnectivity in situations where we can't make all programs read and write super-general formats. [Inside Macintosh chapter "Scrap Manager".] E.g., a graphically-annotated musical score might be supplemented by a stripped down 4-voice melody and by a text (i.e., the lyrics).

The originating program should write the alternate representations in order of "preference": most preferred (most comprehensive) type to least preferred (least comprehensive) type. A receiving program should either use the first appearing type that it understands or search for its own "preferred" type.

A scrap file should have at most one alternative of any type. (A LIST of same type objects is OK as one of the alternatives.) But don't count on this when reading; ignore extra sections of a type. Then a program that reads scrap files can read something out of single purpose files.

Rules for Reader Programs

Here are some notes on building programs that read IFF files. For LIST and PROP work, you should also read up on recursive descent parsers. [See, for example, Compiler Construction, An Advanced Course.]

- The standard is very flexible so many programs can exchange data. This implies a program has to scan the file and react to what's actually there in whatever order it appears. An IFF reader program is a parser.
- For interchange to really work, programs must be willing to do some conversion during read-in. If the data isn't exactly what you expect, say, the raster is smaller than those created by your program, then adjust it. Similarly, your program could crop a large picture, add or drop bitplanes, or create/discard a mask plane. The program should give up gracefully on data that it can't convert.
- If it doesn't start with "FORM", "LIST", or "CAT ", it's not an IFF-85 file.
- For any chunk you encounter, you must recognize its type ID to understand its contents.
- For any FORM chunk you encounter, you must recognize its FORM type ID to understand the contained "local chunks". Even if you don't recognize the FORM type, you can still scan it for nested FORMs, LISTs, and CATs of interest.
- Don't forget to skip the implied pad byte after every odd-length chunk, this is *not* included in the chunk count!
- Chunk types LIST, FORM, PROP, and CAT are generic groups. They always contain a subtype ID followed by chunks.
- Readers ought to handle a CAT of FORMs in a file. You may treat the FORMs like document pages to sequence through, or just use the first FORM.
- Many IFF readers completely skip LISTs. "Fully IFF-conforming" readers are those that handle LISTs, even if just to read the first FORM from a file. If you do look into a LIST, you must process shared properties (in PROP chunks) properly. The idea is to get the correct data or none at all.
- The nicest readers are willing to look into unrecognized FORMs for nested FORM types that they do recognize. For example, a musical score may contain nested instrument descriptions and animation or desktop publishing files may contain still pictures. This extra step is highly recommended.

Note to programmers: Processing PROP chunks is not simple! You'll need some background in interpreters with stack frames. If this is foreign to you, build programs that read/write only one FORM per file. For the more intrepid programmers, the next paragraph summarizes how to process LISTs and PROPs.

Allocate a stack frame for every LIST and FORM you encounter and initialize it by copying the stack frame of the parent LIST or FORM. At the top level, you'll need a stack frame initialized to your program's global defaults. While reading each LIST or FORM, store all encountered properties into the current stack frame. In the example ShowILBM, each stack frame has a place for a bitmap header property ILBM.BMHD and a color map property ILBM.CMAP. When you finally get to the ILBM's BODY chunk, use the property settings accumulated in the current stack frame.

An alternate implementation would just remember PROPs encountered, forgetting each on reaching the end of its scope (the end of the containing LIST). When a FORM XXXX is encountered, scan the chunks in all remembered PROPs XXXX, in order, as if they appeared before the chunks actually in the FORM XXXX. This gets trickier if you read FORMs inside of FORMs.

Rules for Writer Programs

Here are some notes on building programs that write IFF files, which is much easier than reading them.

- An IFF file is a single FORM, LIST, or CAT chunk.
- Any IFF-85 file must start with the 4 characters "FORM", "LIST", or "CAT ", followed by a LONG `ckSize`. There should be no data after the chunk end.
- Chunk types LIST, FORM, PROP, and CAT are generic. They always contain a subtype ID followed by chunks. These three IDs are universally reserved, as are "LIS1" through "LIS9", "FOR1" through "FOR9", "CAT1" through "CAT9", and "   ".
- Don't forget to write a 0 pad byte after each odd-length chunk.
- Do not try to edit a file that you don't know how to create. Programs may look into a file and copy out nested FORMs of types that they recognize, but they should not edit and replace the nested FORMs and not add or remove them. Breaking these rules could make the containing structure inconsistent. You may write a new file containing items you copied, or copied and modified, but don't copy structural parts you don't understand.
- You must adhere to the syntax descriptions in Appendix A. E.g., PROPs may only appear inside LISTs.

There are at least four common techniques for writing an IFF group:

(1) build the data in a file mapped into virtual memory.
(2) build the data in memory blocks and use block I/O.
(3) stream write the data piecemeal and (don't forget!) random access back
to set the group (or FORM) length count.
(4) make a preliminary pass to compute the length count then stream write the data.

Issue: The standard disallows "blind" chunk copying for consistency reasons. Perhaps we can define a ckID convention for chunks that are OK to replicate without knowledge of the contents. Any such chunks would need to be internally consistent, and not be bothered by changed external references.

Issue: Stream-writing an IFF FORM can be inconvenient. With random access files one can write all the chunks then go back to fix up the FORM size. With stream access, the FORM size must be calculated before the file is written. When compression is involved, this can be slow or inconvenient. Perhaps we can define an "END " chunk. The stream writer would use -1 ($FFFFFFFF) as the FORM size. The reader would follow each chunk, when the reader reaches an "END ", it would terminate the last -1 sized chunk. Certain new IFF FORMs could require that readers understand "END ".

7. Standards Committee

The following people contributed to the design of this IFF standard:

Bob "Kodiak" Burns, Commodore-Amiga
R. J. Mical, Commodore-Amiga
Jerry Morrison, Electronic Arts
Greg Riker, Electronic Arts
Steve Shaw, Electronic Arts
Barry Walsh, Commodore-Amiga
Oct, 1988 revision by Bryce Nesbitt, and Carolyn Scheppner, Commodore-Amiga

Appendix A. Reference

Type Definitions

The following C typedefs describe standard IFF structures. Declarations to use in practice will vary with the CPU and compiler. For example, 68000 Lattice C produces efficient comparison code if we define ID as a "LONG". A macro "MakeID" builds these IDs at compile time.

```
/* Standard IFF types, expressed in 68000 Lattice C.    */

typedef unsigned char UBYTE;    /*  8 bits unsigned     */
typedef short WORD;             /* 16 bits signed       */
typedef unsigned short UWORD;   /* 16 bits unsigned     */
typedef long LONG;              /* 32 bits signed       */

typedef char ID[4];             /* 4 chars in ' ' through '~' */

typedef struct {
  ID    ckID;
  LONG  ckSize;                 /* sizeof(ckData)       */
  UBYTE ckData[/* ckSize */];
  } Chunk;


/* ID typedef and builder for 68000 Lattice C. */
typedef LONG ID;                /* 4 chars in ' ' through '~'    */

#define MakeID(a,b,c,d)  ( (a)<<24 | (b)<<16 | (c)<<8 | (d) )

/* Globally reserved IDs. */
#define ID_FORM   MakeID('F','O','R','M')
#define ID_LIST   MakeID('L','I','S','T')
#define ID_PROP   MakeID('P','R','O','P')
#define ID_CAT    MakeID('C','A','T',' ')
#define ID_FILLER MakeID(' ',' ',' ',' ')
```

Syntax Definitions

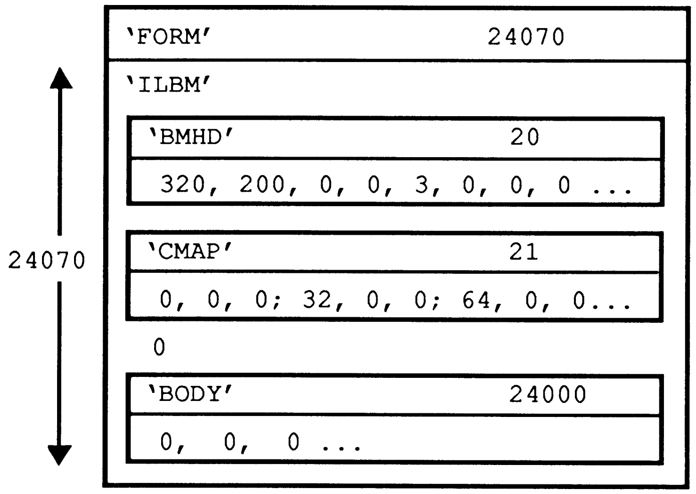
Here's a collection of the syntax definitions in this document.

```
Chunk        ::= ID #{ UBYTE* } [0]

Property     ::= Chunk

FORM         ::= "FORM" #{ FormType (LocalChunk | FORM | LIST | CAT)* }
FormType     ::= ID
LocalChunk   ::= Property | Chunk

CAT          ::= "CAT " #{ ContentsType (FORM | LIST | CAT)* }
ContentsType ::= ID     -- a hint or an "abstract data type" ID

LIST         ::= "LIST" #{ ContentsType PROP* (FORM | LIST | CAT)* }
PROP         ::= "PROP" #{ FormType Property* }
```

In this extended regular expression notation, the token "#" represents a count of the following braced data bytes. Literal items are shown in "quotes", [square bracketed items] are optional, and "*" means 0 or more instances. A sometimes-needed pad byte is shown as "`[0]`".

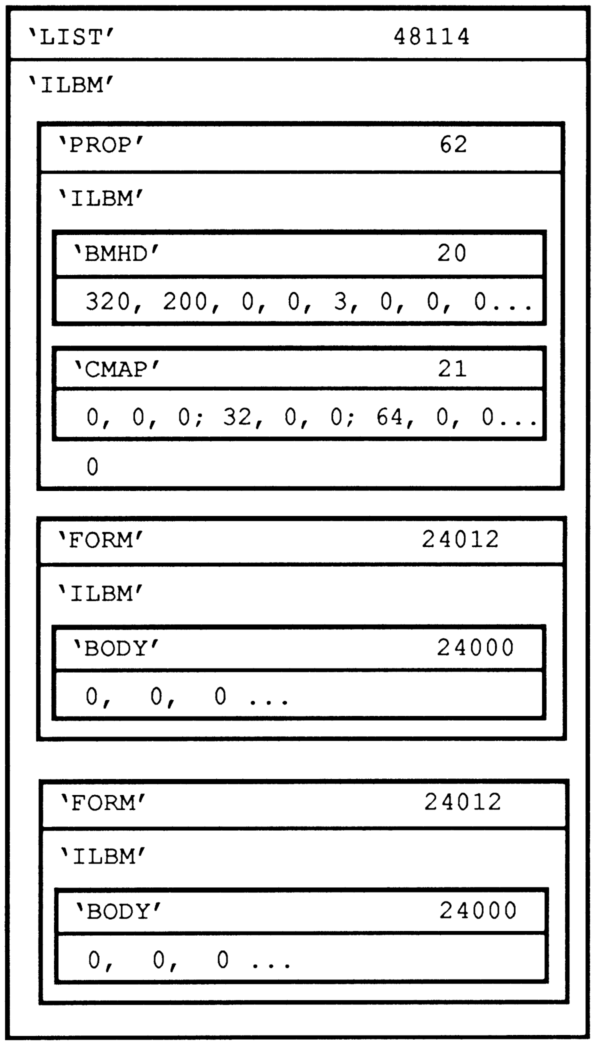
Example Diagrams

Here's a box diagram for an example IFF file, a raster image FORM ILBM. This FORM contains a bitmap header property chunk BMHD, a color map property chunk CMAP, and a raster data chunk BODY. This particular raster is 320 x 200 pixels x 3 bit planes uncompressed. The "`0`" after the CMAP chunk represents a zero pad byte; included since the CMAP chunk has an odd length. The text to the right of the diagram shows the outline that would be printed by the IFFCheck utility program for this particular file.



```
FORM 24070 ILBM

.BMHD 20

.CMAP 21

.BODY 24000
```

This second diagram shows a LIST of two FORMs ILBM sharing a common BMHD property and a common CMAP property. Again, the text on the right is an outline à la *IFFCheck*.



```
LIST 48114 ILBM

.PROP 62 ILBM

.BMHD 20

.CMAP 21

.FORM 24012 ILBM

..BODY 24000

.FORM 24012 ILBM

..BODY 24000
```

"ILBM" IFF Interleaved Bitmap

Date: January 17, 1986 (CRNG data updated Oct, 1988 by Jerry Morrison)
(Appendix E added and CAMG updated Oct, 1988 by Commodore-Amiga, Inc.)
From: Jerry Morrison, Electronic Arts
Status: Released and in use

1. Introduction

"EA IFF 85" is Electronic Arts' standard for interchange format files. "ILBM" is a format for a 2 dimensional raster graphics image, specifically an InterLeaved bitplane BitMap image with color map. An ILBM is an IFF "data section" or "FORM type", which can be an IFF file or a part of one. ILBM allows simple, highly portable raster graphic storage.

An ILBM is an archival representation designed for three uses. First, a stand- alone image that specifies exactly how to display itself (resolution, size, color map, etc.). Second, an image intended to be merged into a bigger picture which has its own depth, color map, and so on. And third, an empty image with a color map selection or "palette" for a paint program. ILBM is also intended as a building block for composite IFF FORMs like "animation sequences" and "structured graphics". Some uses of ILBM will be to preserve as much information as possible across disparate environments. Other uses will be to store data for a single program or highly cooperative programs while maintaining subtle details. So we're trying to accomplish a lot with this one format.

This memo is the IFF supplement for FORM ILBM. Section 2 defines the purpose and format of property chunks bitmap header "BMHD", color map "CMAP", hotspot "GRAB", destination merge data "DEST", sprite information "SPRT", and Commodore Amiga viewport mode "CAMG". Section 3 defines the standard data chunk "BODY". These are the "standard" chunks. Section 4 defines the non- standard data chunks. Additional specialized chunks like texture pattern can be added later. The ILBM syntax is summarized in Appendix A as a regular expression and in Appendix B as a box diagram. Appendix C explains the optional run encoding scheme. Appendix D names the committee responsible for this FORM ILBM standard.

Details of the raster layout are given in part 3, "Standard Data Chunk". Some elements are based on the Commodore Amiga hardware but generalized for use on other computers. An alternative to ILBM would be appropriate for computers with true color data in each pixel, though the wealth of available ILBM images makes import and export important.

Reference:

"EA IFF 85" Standard for Interchange Format Files describes the underlying conventions for all IFF files.
Amiga® is a registered trademark of Commodore-Amiga, Inc.
Electronic Arts™ is a trademark of Electronic Arts.
Macintosh™ is a trademark licensed to Apple Computer, Inc.
MacPaint™ is a trademark of Apple Computer, Inc.

2. Standard Properties

ILBM has several defined property chunks that act on the main data chunks. The required property "BMHD" and any optional properties must appear before any "BODY" chunk. (Since an ILBM has only one BODY chunk, any following properties would be superfluous.) Any of these properties may be shared over a LIST of several IBLMs by putting them in a PROP ILBM (See the EA IFF 85 document).

BMHD

The required property "BMHD" holds a BitmapHeader as defined in the following documentation. It describes the dimensions of the image, the encoding used, and other data necessary to understand the BODY chunk to follow.

```
typedef UBYTE Masking;     /* Choice of masking technique. */

#define mskNone                 0
#define mskHasMask              1
#define mskHasTransparentColor 2
#define mskLasso                3

typedef UBYTE Compression;    /* Choice of compression algorithm
   applied to the rows of all source and mask planes.  "cmpByteRun1"
   is the byte run encoding described in Appendix C.  Do not compress
   across rows! */
#define cmpNone        0
#define cmpByteRun1    1

typedef struct {
  UWORD       w, h;             /* raster width & height in pixels      */
  WORD        x, y;             /* pixel position for this image        */
  UBYTE       nPlanes;          /* # source bitplanes                   */
  Masking     masking;
  Compression compression;
  UBYTE       pad1;             /* unused; ignore on read, write as 0   */
  UWORD       transparentColor; /* transparent "color number" (sort of) */
  UBYTE       xAspect, yAspect; /* pixel aspect, a ratio width : height */
  WORD        pageWidth, pageHeight; /* source "page" size in pixels    */
} BitmapHeader;
```

Fields are filed in the order shown. The UBYTE fields are byte-packed (the C compiler must not add pad bytes to the structure).

The fields `w` and `h` indicate the size of the image rectangle in pixels. Each row of the image is stored in an integral number of 16 bit words. The number of words per row is `words=((w+15)/16)` or `Ceiling(w/16)`. The fields `x` and `y` indicate the desired position of this image within the destination picture. Some reader programs may ignore `x` and `y`. A safe default for writing an ILBM is `(x, y) = (0, 0)`.

The number of source bitplanes in the BODY chunk is stored in `nPlanes`. An ILBM with a CMAP but no BODY and `nPlanes = 0` is the recommended way to store a color map.

Note: Color numbers are color map index values formed by pixels in the destination bitmap, which may be deeper than `nPlanes` if a DEST chunk calls for merging the image into a deeper image.

The field `masking` indicates what kind of masking is to be used for this image. The value `mskNone` designates an opaque rectangular image. The value `mskHasMask` means that a mask plane is interleaved with the bitplanes in the BODY chunk (see below). The value `mskHasTransparentColor` indicates that pixels in the source planes matching `transparentColor` are to be considered "transparent". (Actually, `transparentColor` isn't a "color number" since it's matched with numbers formed by the source bitmap rather than the possibly deeper destination bitmap. Note that having a transparent color implies ignoring one of the color registers. The value mskLasso indicates the reader may construct a mask by lassoing the image as in MacPaint™. To do this, put a 1 pixel border of transparentColor around the image rectangle. Then do a seed fill from this border. Filled pixels are to be transparent.

Issue: Include in an appendix an algorithm for converting a transparent color to a mask plane, and maybe a lasso algorithm.

A code indicating the kind of data compression used is stored in compression. Beware that using data compression makes your data unreadable by programs that don't implement the matching decompression algorithm. So we'll employ as few compression encodings as possible. The run encoding byteRun1 is documented in Appendix C.

The field pad1 is a pad byte reserved for future use. It must be set to 0 for consistency.

The transparentColor specifies which bit pattern means "transparent". This only applies if masking is mskHasTransparentColor or mskLasso. Otherwise, transparentColor should be 0 (see above).

The pixel aspect ratio is stored as a ratio in the two fields xAspect and yAspect. This may be used by programs to compensate for different aspects or to help interpret the fields w, h, x, y, pageWidth, and pageHeight, which are in units of pixels. The fraction xAspect/yAspect represents a pixel's width/height. It's recommended that your programs store proper fractions in the BitmapHeader, but aspect ratios can always be correctly compared with the test:

```
        xAspect * yDesiredAspect = yAspect * xDesiredAspect
```

Typical values for aspect ratio are width : height = 10 : 11 for an Amiga 320 x 200 display and 1 : 1 for a Macintosh™ display.

The size in pixels of the source "page" (any raster device) is stored in pageWidth and pageHeight, e.g., (320, 200) for a low resolution Amiga display. This information might be used to scale an image or to automatically set the display format to suit the image. Note that the image can be larger than the page.

CMAP

The optional (but encouraged) property "CMAP" stores color map data as triplets of red, green, and blue intensity values. The n color map entries ("color registers") are stored in the order 0 through n-1, totaling 3n bytes. Thus n is the ckSize/3. Normally, n would equal $2^{nPlanes}$.

A CMAP chunk contains a ColorMap array as defined below. Note that these typedefs assume a C compiler that implements packed arrays of 3-byte elements.

```
    typedef struct {
        UBYTE red, green, blue;         /* color intensities 0..255 */
        } ColorRegister;                /* size = 3 bytes           */

    typedef ColorRegister ColorMap[n];  /* size = 3n bytes          */
```

The color components red, green, and blue are each stored as a byte (8 bits) representing fractional intensity values expressed in 256ths in the range 0 through 255 (e.g., 24/256). White is (255,255,255—i.e., hex 0xFF,0xFF,0xFF) and black is (0,0,0). If your machine has less color resolution, use the higher order color bits when displaying by simply shifting the CMAP R, G, and B values to the right. When writing a CMAP, storage of less than 8 bits each of R, G, and B was previously accomplished by left justifying the significant bits within the stored bytes (i.e., a 4-bit per gun value of 0xF,0xF,0xF was stored as 0xF0,0xF0,0xF0). This provided correct color values when the ILBM was redisplayed on the same hardware since the zeros were shifted back out.

However, if color values stored by the above method were used as-is when redisplaying on hardware with more color resolution, diminished color could result. For example, a value of (0xF0,0xF0,0xF0) would be pure white on 4-bit-per-gun hardware (i.e., 0xF,0xF,0xF), but not quite white (0xF0,0xF0,0xF0) on 8-bit-per-gun hardware.

Therefore, when storing CMAP values, it is now suggested that you store full 8 bit values for R, G, and B which correctly scale your color values for eight bits. For 4-bit RGB values, this can be as simple as duplicating the 4-bit values in both the upper and lower parts of the bytes—i.e., store (0x1,0x7,0xF) as (0x11,0x77,0xFF). This will provide a more correct color rendition if the image is displayed on a device with 8 bits per gun.

When reading in a CMAP for 8-bit-per-gun display or manipulation, you may want to assume that any CMAP which has 0 values for the low bits of all guns for all registers was stored shifted rather than scaled, and provide your own scaling. Use defaults if the color map is absent or has fewer color registers than you need. Ignore any extra color registers.

The example type `Color4` represents the format of a color register in working memory of an Amiga computer, which has 4 bit video DACs. (The "`:4`" tells smarter C compilers to pack the field into 4 bits.)

```
typedef struct {
    unsigned pad1 :4, red :4, green :4, blue :4;
    } Color4;                    /*   Amiga RAM format.  Not filed.  */
```

Remember that every chunk must be padded to an even length, so a color map with an odd number of entries would be followed by a 0 byte, not included in the `ckSize`.

Storing 24-bit ILBMs Information on storing 24-bit ILBMs can be found in the appendix of this section.

GRAB

The optional property "GRAB" locates a "handle" or "hotspot" of the image relative to its upper left corner, e.g., when used as a mouse cursor or a "paint brush". A GRAB chunk contains a `Point2D`.

```
typedef struct {
    WORD x, y;          /* relative coordinates (pixels) */
    } Point2D;
```

DEST

The optional property "DEST" is a way to say how to scatter zero or more source bitplanes into a deeper destination image. Some readers may ignore DEST.

The contents of a DEST chunk is Destmerge structure:

```
typedef struct {
    UBYTE depth;      /* # bitplanes in the original source            */
    UBYTE pad1;       /* unused; for consistency put 0 here            */
    UWORD planePick;  /* how to scatter source bitplanes into destination */
    UWORD planeOnOff; /* default bitplane data for planePick           */
    UWORD planeMask;  /* selects which bitplanes to store into         */
    } Destmerge;
```

The low order depth number of bits in `planePick`, `planeOnOff`, and `planeMask` correspond one-to-one with destination bitplanes. Bit 0 with bitplane 0, etc. (Any higher order bits should be ignored.) "1" bits in `planePick` mean "put the next source bitplane into this bitplane", so the number of "1" bits should equal `nPlanes`. "0" bits mean "put the corresponding bit from `planeOnOff` into this bitplane". Bits in `planeMask` gate writing to the destination bitplane: "1" bits mean "write to this bitplane" while "0" bits mean "leave this bitplane alone". The normal case (with no DEST property) is equivalent to `planePick = planeMask =` $2^{nPlanes} - 1$.

Remember that color numbers are formed by pixels in the destination bitmap (`depth` planes deep) not in the source bitmap (`nPlanes` planes deep).

SPRT

The presence of an "SPRT" chunk indicates that this image is intended as a sprite. It's up to the reader program to actually make it a sprite, if even possible, and to use or overrule the sprite precedence data inside the SPRT chunk:

```
typedef UWORD SpritePrecedence; /* relative precedence, 0 is the highest */
```

Precedence 0 is the highest, denoting a sprite that is foremost.

Creating a sprite may imply other setup. E.g., a 2 plane Amiga sprite would have `transparent-Color` = 0. Color registers 1, 2, and 3 in the CMAP would be stored into the correct hardware color registers for the hardware sprite number used, while CMAP color register 0 would be ignored.

CAMG

A "CAMG" chunk is specifically for Commodore Amiga ILBMs. All Amiga-based reader and writer software should deal with CAMG. The Amiga supports many different video display modes including interlace, Extra Halfbrite, hold and modify (HAM), plus a variety of new modes under the 2.0 operating system. A CAMG chunk contains a single long word (length=4) which specifies the Amiga display mode of the picture.

Prior to 2.0, it was possible to express all available Amiga `ViewModes` in 16 bits of flags (`Viewport->Modes or NewScreen->ViewModes`). Old-style writers and readers place a 16-bit Amiga `ViewModes` value in the low word of the CAMG, and zeros in the high word. The following `Viewmode` flags should always be removed from old-style 16-bit `ViewModes` values when writing or reading them:

```
EXTENDED_MODE | SPRITES | VP_HIDE | GENLOCK_AUDIO | GENLOCK_VIDEO (=0x7102, mask=0x8EFD)
```

New ILBM readers and writers, should treat the full CAMG longword as a 32-bit `ModeID` to support new and future display modes.

New ILBM writers, when running under the 2.0 Amiga operating system, should directly store the full 32-bit return value of the graphics function `GetVPModeID(vp)` in the CAMG longword. When running under 1.3, store a 16-bit `Viewmodes` value masked as described above.

ILBM readers should only mask bits out of a CAMG if the CAMG has a zero upper word (see exception below). New ILBM readers, when running under 2.0, should then treat the 32-bit CAMG value as a `ModeID`, and should use the graphics `ModeNotAvailable()` function to determine if the mode is available. If the mode is not available, fall back to another suitable display mode. When running under 1.3, the low word of the CAMG may generally be used to open a compatible display.

Note that one popular graphics package stores garbage in the upper word of the CAMG of brushes, and incorrect values (generally zero) in the low word. You can screen for such garbage values by testing for non-zero in the upper word of a `ModeID` in conjunction with the 0x00001000 bit NOT set in the low word.

The following code fragment demonstrates ILBM reader filtering of inappropriate bits in 16-bit CAMG values.

```
#include <graphics/view.h>
#include <graphics/displayinfo.h>

/* Knock bad bits out of old-style CAMG modes before checking availability.
 * (some ILBM CAMG's have these bits set in old 1.3 modes, and should not)
 * If not an extended monitor ID, or if marked as extended but missing
 * upper 16 bits, screen out inappropriate bits now.
 */
if((!(modeid & MONITOR_ID_MASK)) ||
    ((modeid & EXTENDED_MODE)&&(!(modeid & 0xFFFF0000))))
modeid &=
    (~(EXTENDED_MODE|SPRITES|GENLOCK_AUDIO|GENLOCK_VIDEO|VP_HIDE));

/* Check for bogus CAMG like some brushes have, with junk in
 * upper word and extended bit NOT set not set in lower word.
 */
if((modeid & 0xFFFF0000)&&(!(modeid & EXTENDED_MODE)))
    {
    /* Bad CAMG, so ignore CAMG and determine a mode based on
     * based on pagesize or aspect
     */
     modeid = NULL;
     if(wide >= 640) modeid |= HIRES;
     if(high >= 400) modeid |= LACE;
     }

/* Now, ModeNotAvailable() may be used to determine if the mode is available.
 *
 * If the mode is not available, you may prompt the user for a mode
 * choice, or search the 2.0 display database for an appropriate
 * replacement mode, or you may be able to get a relatively compatible
 * old display mode by masking out all bits except
 * HIRES | LACE | HAM | EXTRA_HALFBRITE
 */
```

3. Standard "BODY" Data Chunk

Raster Layout

Raster scan proceeds left-to-right (increasing X) across scan lines, then top-to-bottom (increasing Y) down columns of scan lines. The coordinate system is in units of pixels, where (0,0) is the upper left corner.

The raster is typically organized as bitplanes in memory. The corresponding bits from each plane, taken together, make up an index into the color map which gives a color value for that pixel. The first bitplane, plane 0, is the low order bit of these color indexes.

A scan line is made of one "row" from each bitplane. A row is one plane's bits for one scan line, but padded out to a word (2 byte) boundary (not necessarily the first word boundary). Within each row, successive bytes are displayed in order and the most significant bit of each byte is displayed first.

A "mask" is an optional "plane" of data the same size `(w, h)` as a bitplane. It tells how to "cut out" part of the image when painting it onto another image. "One" bits in the mask mean "copy the corresponding pixel to the destination". "Zero" mask bits mean "leave this destination pixel alone". In other words, "zero" bits designate transparent pixels.

The rows of the different bitplanes and mask are interleaved in the file (see below). This localizes all the information pertinent to each scan line. It makes it much easier to transform the data while reading it to adjust the image size or depth. It also makes it possible to scroll a big image by swapping rows directly from the file without the need for random-access to all the bitplanes.

BODY

The source raster is stored in a "BODY" chunk. This one chunk holds all bitplanes and the optional mask, interleaved by row.

The BitMapHeader, in a BMHD property chunk, specifies the raster's dimensions `w`, `h`, and `nPlanes`. It also holds the `masking` field which indicates if there is a mask plane and the `compression` field which indicates the compression algorithm used. This information is needed to interpret the BODY chunk, so the BMHD chunk must appear first. While reading an ILBM's BODY, a program may convert the image to another size by filling (with `transparentColor`) or clipping.

The BODY's content is a concatenation of scan lines. Each scan line is a concatenation of one row of data from each plane in order `0` through `nPlanes-1` followed by one row from the mask (if `masking = hasMask`). If the `BitMapHeader` field compression is `cmpNone`, all `h` rows are exactly `(w+15)/16` words wide. Otherwise, every row is compressed according to the specified algorithm and the stored widths depend on the data compression.

Reader programs that require fewer bitplanes than appear in a particular ILBM file can combine planes or drop the high-order (later) planes. Similarly, they may add bitplanes and/or discard the mask plane.

Do not compress across rows, and don't forget to compress the mask just like the bitplanes. Remember to pad any BODY chunk that contains an odd number of bytes and skip the pad when reading.

4. Nonstandard Data Chunks

The following data chunks were defined after various programs began using FORM ILBM so they are "nonstandard" chunks. See the registry document for the latest information on additional non-standard chunks.

CRNG

A "CRNG" chunk contains "color register range" information. It's used by Electronic Arts' Deluxe Paint program to identify a contiguous range of color registers for a "shade range" and color cycling. There can be zero or more CRNG chunks in an ILBM, but all should appear before the BODY chunk. Deluxe Paint normally writes 4 CRNG chunks in an ILBM when the user asks it to "Save Picture".

```
typedef struct {
    WORD  pad1;       /* reserved for future use; store 0 here    */
    WORD  rate;       /* color cycle rate                         */
    WORD  flags;      /* see below                                */
    UBYTE low, high;  /* lower and upper color registers selected */
    } CRange;
```

The bits of the `flags` word are interpreted as follows: if the low bit is set then the cycle is "active", and if this bit is clear it is not active. Normally, color cycling is done so that colors move to the next higher position in the cycle, with the color in the high slot moving around to the low slot. If the second bit of the flags word is set, the cycle moves in the opposite direction. As usual, the other bits of the flags word are reserved for future expansion. Here are the masks to test these bits:

```
#define RNG_ACTIVE  1
#define RNG_REVERSE 2
```

The fields `low` and `high` indicate the range of color registers (color numbers) selected by this `CRange`.

The field `active` indicates whether color cycling is on or off. Zero means off.

The field `rate` determines the speed at which the colors will step when color cycling is on. The units are such that a rate of 60 steps per second is represented as 2^{14} = 16384. Slower rates can be obtained by linear scaling: for 30 steps/second, rate = 8192; for 1 step/second, rate = 16384/60 $\approx$ 273.

Warning! One popular paint package always sets the RNG_ACTIVE bit, but uses a rate of 36 (decimal) to indicate cycling is not active.

CCRT

Commodore's Graphicraft program uses a similar chunk "CCRT" (for Color Cycling Range and Timing). This chunk contains a `CycleInfo` structure.

```
typedef struct {
    WORD  direction;     /* 0 = don't cycle.  1 = cycle forwards      */
                         /* (1, 2, 3). -1 = cycle backwards (3, 2, 1) */
    UBYTE start, end;    /* lower and upper color registers selected  */
    LONG  seconds;       /* # seconds between changing colors plus... */
    LONG  microseconds;  /* # microseconds between changing colors    */
    WORD  pad;           /* reserved for future use; store 0 here     */
    } CycleInfo;
```

This is very similar to a CRNG chunk. A program would probably only use one of these two methods of expressing color cycle data, new programs should use CRNG. You could write out both if you want to communicate this information to both Deluxe Paint and Graphicraft.

Appendix A. ILBM Regular Expression

Here's a regular expression summary of the FORM ILBM syntax. This could be an IFF file or a part of one.

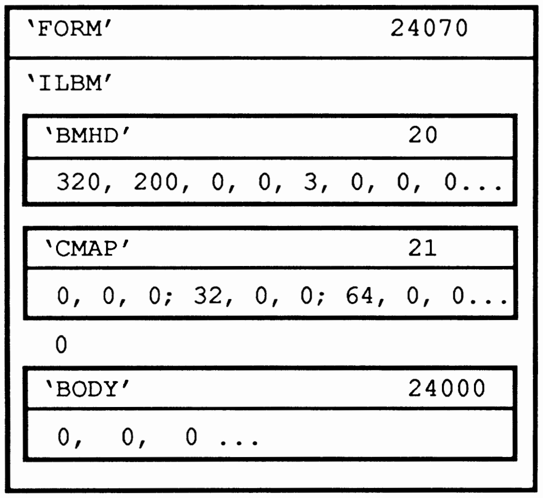
```
ILBM ::= "FORM" #{    "ILBM" BMHD [CMAP] [GRAB] [DEST] [SPRT] [CAMG]
                        CRNG* CCRT* [BODY]    }

BMHD ::= "BMHD" #{    BitMapHeader    }
CMAP ::= "CMAP" #{    (red green blue)*    } [0]
GRAB ::= "GRAB" #{    Point2D    }
DEST ::= "DEST" #{    DestMerge    }
SPRT ::= "SPRT" #{    SpritePrecedence    }
CAMG ::= "CAMG" #{    LONG    }

CRNG ::= "CRNG" #{    CRange    }
CCRT ::= "CCRT" #{    CycleInfo    }
BODY ::= "BODY" #{    UBYTE*    } [0]
```

The token "#" represents a `ckSize` LONG count of the following braced data bytes. E.g., a BMHD's "#" should equal `sizeof(BitMapHeader)`. Literal strings are shown in "quotes", [square bracket items] are optional, and "*" means 0 or more repetitions. A sometimes-needed pad byte is shown as "`[0]`".

The property chunks BMHD, CMAP, GRAB, DEST, SPRT, CAMG and any CRNG and CCRT data chunks may actually be in any order but all must appear before the BODY chunk since ILBM readers usually stop as soon as they read the BODY. If any of the 6 property chunks are missing, default values are inherited from any shared properties (if the ILBM appears inside an IFF LIST with PROPs) or from the reader program's defaults. If any property appears more than once, the last occurrence before the BODY is the one that counts since that's the one that modifies the BODY.

Appendix B. ILBM Box Diagram

Here is a box diagram for a simple example: an uncompressed image 320 x 200 pixels x 3 bitplanes. The text to the right of the diagram shows the outline that would be printed by the Sift utility program for this particular file.



```
FORM 24070 ILBM

.BMHD 20

.CMAP 21

.BODY 24000
```

The "`0`" after the CMAP chunk is a pad byte.

Appendix C. IFF Hints

Hints on ILBM files from Jerry Morrison, Oct 1988. How to avoid some pitfalls when reading ILBM files:

- Don't ignore the BitMapHeader.masking field. A bitmap with a mask (such as a partially-transparent DPaint brush or a DPaint picture with a stencil) will read as garbage if you don't de-interleave the mask.
- Don't assume all images are compressed. Narrow images aren't usually run-compressed since that would actually make them longer.
- Don't assume a particular image size. You may encounter overscan pictures and PAL pictures.

Different hardware display devices have different color resolutions:

Device	R:G:B bits	maxColor
Mac SE	1	1
IBM EGA	2:2:2	3
Atari ST	3:3:3	7
Amiga	4:4:4	15
CD-I	5:5:5	31
IBM VGA	6:6:6	63
Mac II	8:8:8	255

An ILBM CMAP defines 8 bits of Red, Green and Blue (i.e., 8:8:8 bits of R:G:B). When displaying on hardware which has less color resolution, just take the high order bits. For example, to convert ILBM's 8-bit Red to the Amiga's 4-bit Red, right shift the data by 4 bits (R4 := R8 >> 4).

To convert hardware colors to ILBM colors, the ILBM specification says just set the high bits (R8 := R4 << 4). But you can transmit higher contrast to foreign display devices by scaling the data [0..maxColor] to the full range [0..255]. In other words, R8 := (Rn x 255) ? maxColor. (Example #1: EGA color 1:2:3 scales to 85:170:255. Example #2: Amiga 15:7:0 scales to 255:119:0). This makes a big difference where maxColor is less than 15. In the extreme case, Mac SE white (1) should be converted to ILBM white (255), not to ILBM gray (128).

CGA and EGA subtleties

IBM EGA colors in 350 scan line mode are 2:2:2 bits of R:G:B, stored in memory as xxR'G'B'RBG. That's 3 low-order bits followed by 3 high-order bits.

IBM CGA colors are 4 bits stored in a byte as xxxxIRGB. (EGA colors in 200 scan line modes are the same as CGA colors, but stored in memory as xxxIxRGB.) That's 3 high-order bits (one for each of R, G, and B) plus one low-order " Intensity" bit for all 3 components R, G, and B. Exception: IBM monitors show IRGB = 0110 as brown, which is really the EGA color R:G:B = 2:1:0, not dark yellow 2:2:0.

24-bit ILBMs

When storing deep images as ILBMs (e.g., images with 8 bits each of R,G, and B), the bits for each pixel represent an absolute RGB value for that pixel rather than an index into a limited color map. The order for saving the bits is critical since a deep ILBM would not contain the usual CMAP of RGB values (such a CMAP would be too large and redundant).

To interpret these "deep" ILBMs, it is necessary to have a standard order in which the bits of the R, G, and B values will be stored. A number of different orderings have already been used in deep ILBMs and a default has been chosen from them.

The following bit ordering has been chosen as the default bit ordering for deep ILBMs.

Default standard deep ILBM bit ordering:
saved first——————————————————————→saved last
R0 R1 R2 R3 R4 R5 R6 R7 G0 G1 G2 G3 G4 G5 G6 G7 B0 B1 B2 B3 B4 B5 B6 B7

One other existing deep bit ordering that you may encounter is the 21-bit NewTek format.

NewTek deep ILBM bit ordering:
saved first——————————————————————→saved last
R7 G7 B7 R6 G6 B6 R5 G5 B5 R4 G4 B4 R3 G3 B3 R2 G2 B2 R1 G1 B1 R0 G0 B0

Note that you may encounter CLUT chunks in deep ILBMs. See the Third Party Specs appendix for more information on CLUT chunks.

Appendix D. ByteRun1 Run Encoding

The run encoding scheme `byteRun1` is best described by pseudo code for the decoder `Unpacker` (called `UnPackBits` in the Macintosh™ toolbox):

```
UnPacker:
  LOOP until produced the desired number of bytes
      Read the next source byte into n
      SELECT n FROM
          [0..127]    => copy the next n+1 bytes literally
          [-1..-127]  => replicate the next byte -n+1 times
          -128        => no operation
          ENDCASE;
      ENDLOOP;
```

In the inverse routine `Packer`, it's best to encode a 2 byte repeat run as a replicate run except when preceded and followed by a literal run, in which case it's best to merge the three into one literal run. Always encode 3 byte repeats as replicate runs.

Remember that each row of each scan line of a raster is separately packed.

Appendix E. Standards Committee

The following people contributed to the design of this FORM ILBM standard:

Bob "Kodiak" Burns, Commodore-Amiga
R. J. Mical, Commodore-Amiga
Jerry Morrison, Electronic Arts
Greg Riker, Electronic Arts
Steve Shaw, Electronic Arts
Dan Silva, Electronic Arts
Barry Walsh, Commodore-Amiga

"FTXT" IFF Formatted Text

Date: November 15, 1985 (Updated Oct, 1988 Commodore-Amiga, Inc.)
From: Steve Shaw and Jerry Morrison, Electronic Arts and Bob "Kodiak" Burns, Commodore-Amiga
Status: Adopted

1. Introduction

This memo is the IFF supplement for FORM FTXT. An FTXT is an IFF "data section" or "FORM type"—which can be an IFF file or a part of one—containing a stream of text plus optional formatting information."EA IFF 85" is Electronic Arts' standard for interchange format files. (See the IFF reference.)

An FTXT is an archival and interchange representation designed for three uses. The simplest use is for a "console device" or "glass teletype" (the minimal 2-D text layout means): a stream of "graphic" ("printable") characters plus positioning characters "space" ("SP") and line terminator ("LF"). This is not intended for cursor movements on a screen although it does not conflict with standard cursor-moving characters. The second use is text that has explicit formatting information (or "looks") such as font family and size, typeface, etc. The third use is as the lowest layer of a structured document that also has "inherited" styles to implicitly control character looks. For that use, FORMs FTXT would be embedded within a future document FORM type. The beauty of FTXT is that these three uses are interchangeable, that is, a program written for one purpose can read and write the others' files. So a word processor does not have to write a separate plain text file to communicate with other programs.

Text is stored in one or more "CHRS" chunks inside an FTXT. Each CHRS contains a stream of 8-bit text compatible with ISO and ANSI data interchange standards. FTXT uses just the central character set from the ISO/ANSI standards. (These two standards are henceforth called "ISO/ANSI" as in "see the ISO/ANSI reference".)

Since it's possible to extract just the text portions from future document FORM types, programs can exchange data without having to save both plain text and formatted text representations.

Character looks are stored as embedded control sequences within CHRS chunks. This document specifies which class of control sequences to use: the CSI group. This document does not yet specify their meanings, e.g., which one means "turn on italic face". Consult ISO/ANSI.

Section 2 defines the chunk types character stream "CHRS" and font specifier "FONS". These are the "standard" chunks. Specialized chunks for private or future needs can be added later. Section 3 outlines an FTXT reader program that strips a document down to plain unformatted text. Appendix A is a code table for the 8-bit ISO/ANSI character set used here. Appendix B is an example FTXT shown as a box diagram. Appendix C is a racetrack diagram of the syntax of ISO/ANSI control sequences.

Reference:

Amiga® is a registered trademark of Commodore-Amiga, Inc. Electronic Arts™ is a trademark of Electronic Arts.

IFF: <u>"EA IFF 85" Standard for Interchange Format Files</u> describes the underlying conventions for all IFF files.

ISO/ANSI: ISO/DIS 6429.2 and ANSI X3.64-1979. International Organization for Standardization (ISO) and American National Standards Institute (ANSI) data-interchange standards. The relevant parts of these two standards documents are identical. ISO standard 2022 is also relevant.

2. Standard Data and Property Chunks

The main contents of a FORM FTXT is in its character stream "CHRS" chunks. Formatting property chunks may also appear. The only formatting property yet defined is "FONS", a font specifier. A FORM FTXT with no CHRS represents an empty text stream. A FORM FTXT may contain nested IFF FORMs, LISTs, or CATs, although a "stripping" reader (see section 3) will ignore them.

Character Set

FORM FTXT uses the core of the 8-bit character set defined by the ISO/ANSI standards cited at the start of this document. (See Appendix A for a character code table.) This character set is divided into two "graphic" groups plus two "control" groups. Eight of the control characters begin ISO/ANSI standard control sequences. (See "Control Sequences" below.) Most control sequences and control characters are reserved for future use and for compatibility with ISO/ANSI. Current reader programs should skip them.

- C0 is the group of control characters in the range NUL (hex 0) through hex 1F. Of these, only LF (hex 0A) and ESC (hex 1B) are significant. ESC begins a control sequence. LF is the line terminator, meaning "go to the first horizontal position of the next line". All other C0 characters are not used. In particular, CR (hex 0D) is not recognized as a line terminator.
- G0 is the group of graphic characters in the range hex 20 through hex 7F. SP (hex 20) is the space character. DEL (hex 7F) is the delete character which is not used. The rest are the standard ASCII printable characters "!" (hex 21) through "~" (hex 7E).
- C1 is the group of extended control characters in the range hex 80 through hex 9F. Some of these begin control sequences. The control sequence starting with CSI (hex 9B) is used for FTXT formatting. All other control sequences and C1 control characters are unused.
- G1 is the group of extended graphic characters in the range NBSP (hex A0) through "ÿ" (hex FF). It is one of the alternate graphic groups proposed for ISO/ANSI standardization.

Control Sequences

Eight of the control characters begin ISO/ANSI standard "control sequences" (or "escape sequences"). These sequences are described below and diagramed in Appendix C.

```
G0         ::= (SP through DEL)
G1         ::= (NBSP through "ÿ")

ESC-Seq    ::= ESC (SP through "/") * ("0" through "~")
ShiftToG2  ::= SS2 G0
ShiftToG3  ::= SS3 G0
CSI-Seq    ::= CSI (SP through "?") * ("@" through "~")
DCS-Seq    ::= (DCS | OSC | PM | APC) (SP through "~" | G1) * ST
```

"ESC-Seq" is the control sequence ESC (hex 1B), followed by zero or more characters in the range SP through "/" (hex 20 through hex 2F), followed by a character in the range "0" through "~" (hex 30 through hex 7E). These sequences are reserved for future use and should be skipped by current FTXT reader programs.

SS2 (hex 8E) and SS3 (hex 8F) shift the single following G0 character into yet-to-be-defined graphic sets G2 and G3, respectively. These sequences should not be used until the character sets G2 and G3 are standardized. A reader may simply skip the SS2 or SS3 (taking the following character as a corresponding G0 character) or replace the two-character sequence with a character like "?" to mean "absent".

FTXT uses "CSI-Seq" control sequences to store character formatting (font selection by number, type face, and text size) and perhaps layout information (position and rotation). "CSI-Seq" control sequences start with CSI (the "control sequence introducer", hex 9B). Syntactically, the sequence includes zero or more characters in the range SP through "?" (hex 20 through hex 3F) and a concluding character in the range "@" through "~" (hex 40 through hex 7E). These sequences may be skipped by a minimal FTXT reader, i.e., one that ignores formatting information.

Note: A future FTXT standardization document will explain the uses of CSI-Seq sequences for setting character face (light weight vs. medium vs. bold, italic vs. upright, height, pitch, position, and rotation). For now, consult the ISO/ANSI references.

"DCS-Seq" is the control sequences starting with DCS (hex 90), OSC (hex 9D), PM (hex 9E), or APC (hex 9F), followed by zero or more characters each of which is in the range SP through "~" (hex 20 through hex 7E) or else a G1 character, and terminated by an ST (hex 9C). These sequences are reserved for future use and should be skipped by current FTXT reader programs.

Data Chunk CHRS

A CHRS chunk contains a sequence of 8-bit characters abiding by the ISO/ANSI standards cited at the start of this document. This includes the character set and control sequences as described above and summarized in Appendix A and C.

A FORM FTXT may contain any number of CHRS chunks. Taken together, they represent a single stream of textual information. That is, the contents of CHRS chunks are effectively concatenated except that (1) each control sequence must be completely within a single CHRS chunk, and (2) any formatting property chunks appearing between two CHRS chunks affects the formatting of the latter chunk's text. Any formatting settings set by control sequences inside a CHRS carry over to the next CHRS in the same FORM FTXT. All formatting properties stop at the end of the FORM since IFF specifies that adjacent FORMs are independent of each other (although not independent of any properties inherited from an enclosing LIST or FORM).

Property Chunk FONS

The optional property "FONS" holds a FontSpecifier as defined in the C declaration below. It assigns a font to a numbered "font register" so it can be referenced by number within subsequent CHRS chunks. (This function is not provided within the ISO and ANSI standards.) The font specifier gives both a name and a description for the font so the recipient program can do font substitution.

By default, CHRS text uses font 1 until it selects another font. A minimal text reader always uses font 1. If font 1 hasn't been specified, the reader may use the local system font as font 1.

```
typedef struct {
  UBYTE id;      /* 0 through 9 is a font id number referenced by an SGR
                    control sequence selective parameter of 10 through 19.
                    Other values are reserved for future standardization. */
  UBYTE pad1;    /* reserved for future use; store 0 here                  */
  UBYTE proportional; /* proportional font-- 0=unknown, 1=no, 2=yes        */
  UBYTE serif;   /* serif font-- 0 = unknown, 1 = no, 2 = yes              */
  char  name[]; /* A NUL-terminated string naming the preferred font.     */
  } FontSpecifier;
```

Fields are filed in the order shown. The UBYTE fields are byte-packed (2 per 16-bit word). The field `pad1` is reserved for future standardization. Programs should store 0 there for now.

The field `proportional` indicates if the desired font is proportional width as opposed to fixed width. The field `serif` indicates if the desired font is serif as opposed to sans serif. Issue: Discuss font substitution!

Future Properties

New optional property chunks may be defined in the future to store additional formatting information. They will be used to represent formatting not encoded in standard ISO/ANSI control sequences and for "inherited" formatting in structured documents. Text orientation might be one example.

Positioning Units

Unless otherwise specified, position and size units used in FTXT formatting properties and control sequences are in decipoints (720 decipoints/inch). This is ANSI/ISO Positioning Unit Mode (PUM) 2. While a metric standard might be nice, decipoints allow the existing U.S.A. typographic units to be encoded easily, e.g., "12 points" is "120 decipoints".

3. FTXT Stripper

An FTXT reader program can read the text and ignore all formatting and structural information in a document FORM that uses FORMs FTXT for the leaf nodes. This amounts to stripping a document down to a stream of plain text. It would do this by skipping over all chunks except FTXT.CHRS (CHRS chunks found inside a FORM FTXT) and within the FTXT.CHRS chunks skipping all control characters and control sequences. (Appendix C diagrams this text scanner.) It may also read FTXT.FONS chunks to find a description for font 1.

Appendix A: Character Code Table

This table corresponds to the ISO/DIS 6429.2 and ANSI X3.64-1979 8-bit character set standards. Only the core character set of those standards is used in FTXT.

Two G1 characters aren't defined in the standards and are shown as dark gray entries in this table. Light gray shading denotes control characters. (DEL is a control character although it belongs to the graphic group G0.)

ISO / DIS 6429.2 and ANSI X3.64-1979 Character Code Table

MSN (most significant nibble)

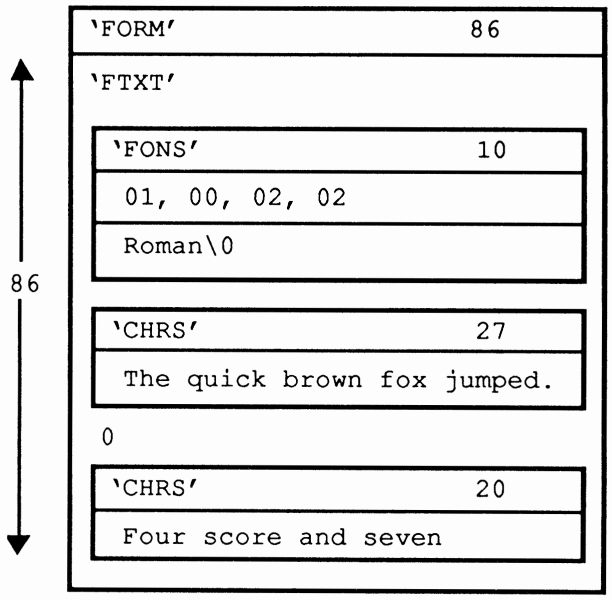
LSN	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL		SP	0	@	P	`	p		DCS	NBSP	°	À		à	
1			!	1	A	Q	a	q			¡		Á	Ñ	á	ñ
2			"	2	B	R	b	r			¢		Â	Ò	â	ò
3			#	3	C	S	c	s			£		Ã	Ó	ã	ó
4			$	4	D	T	d	t			¤		Ä	Ô	ä	ô
5			%	5	E	U	e	u			¥		Å	Õ	å	õ
6			&	6	F	V	f	v				¶	Æ	Ö	æ	ö
7			'	7	G	W	g	w			§	•	Ç		ç	
8			(	8	H	X	h	x					È	Ø	è	ø
9			)	9	I	Y	i	y			©		É	Ù	é	ù
A	LF		*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
B		ESC	+	;	K	[	k	{		CSI	«	»	Ë	Û	ë	û
C			,	<	L		l	|		ST	¬		Ì	Ü	ì	ü
D	CR		-	=	M	]	m	}		OSC	SHY		Í		í	
E			.	>	N	^	n	~	SS2	PM	®		Î		î	
F			/	?	O	_	o	DEL	SS3	APC	¯	¿	Ï	ß	ï	ÿ

Control group C0 (columns 0–1) · Graphic group G0 (columns 2–7) · Control group C1 (columns 8–9) · Graphic group G1 (columns A–F)

NBSP is a non-breaking space
SHY is a soft hyphen

Appendix B. FTXT Example

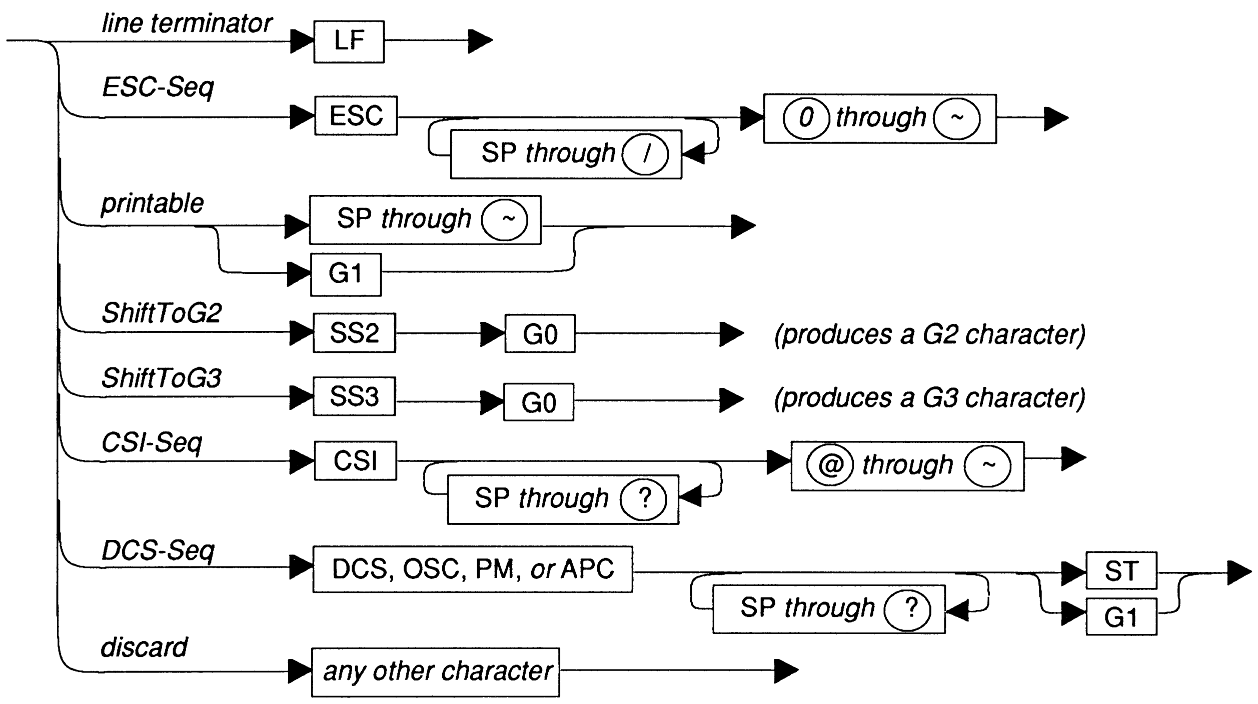
Here's a box diagram for a simple example: "The quick brown fox jumped. Four score and seven", written in a proportional serif font named "Roman".



The "0" after the first CHRS chunk is a pad byte.

Appendix C. ISO/ANSI Control Sequences

This is a racetrack diagram of the ISO/ANSI characters and control sequences as used in FTXT.CHRS chunks.



Of the various control sequences, only CSI-Seq is used for FTXT character formatting information. The others are reserved for future use and for compatibility with ISO/ANSI standards. Certain character sequences are syntactically malformed, e.g., CSI followed by a C0, C1, or G1 character. Writer programs should not generate reserved or malformed sequences and reader programs should skip them.

Consult the ISO/ANSI standards for the meaning of the CSI-Seq control sequences.

The two character set shifts SS2 and SS3 may be used when the graphic character groups G2 and G3 become standardized.

"SMUS" IFF Simple Musical Score

Date: February 20, 1987 (SID _Clef and SID_Tempo added Oct, 1988)
From: Jerry Morrison, Electronic Arts
Status: Adopted

1. Introduction

This is a reference manual for the data interchange format "SMUS", which stands for Simple MUsical Score. "EA IFF 85" is Electronic Arts' standard for interchange format files. A FORM or "data section") such as FORM SMUS can be an IFF file or a part of one. [See "EA IFF 85" Electronic Arts Interchange File Format.]

SMUS is a practical data format for uses like moving limited scores between programs and storing theme songs for game programs. The format should be geared for easy read-in and playback. So FORM SMUS uses the compact time encoding of Common Music Notation (half notes, dotted quarter rests, etc.). The SMUS format should also be structurally simple. So it has no provisions for fancy notational information needed by graphical score editors or the more general timing (overlapping notes, etc.) and continuous data (pitch bends, etc.) needed by performance-oriented MIDI recorders and sequencers. Complex music programs may wish to save in a more complete format, but still import and export SMUS when requested.

A SMUS score can say which "instruments" are supposed play which notes. But the score is independent of whatever output device and driver software is used to perform the notes. The score can contain device- and driver-dependent instrument data, but this is just a cache. As long as a SMUS file stays in one environment, the embedded instrument data is very convenient. When you move a SMUS file between programs or hardware configurations, the contents of this cache usually become useless.

Like all IFF formats, SMUS is a filed or "archive" format. It is completely independent of score representations in working memory, editing operations, user interface, display graphics, computation hardware, and sound hardware. Like all IFF formats, SMUS is extensible.

SMUS is not an end-all musical score format. Other formats may be more appropriate for certain uses. (We'd like to design an general-use IFF score format "GSCR". FORM GSCR would encode fancy notational data and performance data. There would be a SMUS to/from GSCR converter.)

Section 2 gives important background information. Section 3 details the SMUS components by defining the required property score header "SHDR", the optional text properties name "NAME", copyright "(c) ", and author "AUTH", optional text annotation "ANNO", the optional instrument specifier "INS1", and the track data chunk "TRAK". Section 4 defines some chunks for particular programs to store private information. These are "standard" chunks; specialized chunks for future needs can be added later. Appendix A is a quick-reference summary. Appendix B is an example box diagram. Appendix C names the committee responsible for this standard.

References:

"EA IFF 85" Standard for Interchange Format Files describes the underlying conventions for all IFF files.
"8SVX" IFF 8-Bit Sampled Voice documents a data format for sampled instruments.
MIDI: Musical Instrument Digital Interface Specification 1.0, International MIDI Association, 1983.

SSSP: See various articles on Structured Sound Synthesis Project in Foundations of Computer Music.

Electronic Arts^{TM} is a trademark of Electronic Arts.
Amiga® is a registered trademark of Commodore-Amiga, Inc.

2. Background

Here's some background information on score representation in general and design choices for SMUS.

First, we'll borrow some terminology from the Structured Sound Synthesis Project. [See the SSSP reference.] A "musical note" is one kind of *scheduled event*. Its properties include an *event duration*, an *event delay*, and a *timbre object*. The *event duration* tells the scheduler how long the note should last. The *event delay* tells how long after starting this note to wait before starting the next event. The *timbre object* selects sound driver data for the note; an "instrument" or "timbre". A "rest" is a sort of a null event. Its only property is an event delay.

Classical Event Durations

SMUS is geared for "classical" scores, not free-form performances. So its event durations are classical (whole note, dotted quarter rest, etc.). SMUS can tie notes together to build a "note event" with an unusual event duration. The set of useful classical durations is very small. So SMUS needs only a handful of bits to encode an event duration. This is very compact. It's also very easy to display in Common Music Notation (CMN).

Tracks

The events in a SMUS score are grouped into parallel "tracks". Each track is a linear stream of events.

Why use tracks? Tracks serve 4 functions:

1. Tracks make it possible to encode event delays very compactly. A "classical" score has chorded notes and sequential notes; no overlapping notes. That is, each event begins either simultaneous with or immediately following the previous event in that track. So each event delay is either 0 or the same as the event's duration. This binary distinction requires only one bit of storage.
2. Tracks represent the "voice tracks" in Common Music Notation. CMN organizes a score in parallel staves, with one or two "voice tracks" per staff. So one or two SMUS tracks represents a CMN staff.
3. Tracks are a good match to available sound hardware. We can use "instrument settings" in a track to store the timbre assignments for that track's notes. The instrument setting may change over the track.
4. Furthermore, tracks can help to allocate notes among available output channels or performance devices or tape recorder "tracks". Tracks can also help to adapt polyphonic data to monophonic output channels.

5. Tracks are a good match to simple sound software. Each track is a place to hold state settings like "dynamic mark *pp* ", "time signature 3/4", "mute this track", etc., just as it's a context for instrument settings. This is a lot like a text stream with running "font" and "face" properties (attributes). Running state is usually more compact than, say, storing an instrument setting in every note event. It's also a useful way to organize "attributes" of notes. With "running track state" we can define new note attributes in an upward- and backward-compatible way.

   Running track state can be expanded (run decoded) while loading a track into memory or while playing the track. The runtime track state must be reinitialized every time the score is played.

Separated vs. interleaved tracks. Multi-track data could be stored either as separate event streams or interleaved into one stream. To interleave the streams, each event has to carry a "track number" attribute.

If we were designing an editable score format, we might interleave the streams so that nearby events are stored nearby. This helps when searching the data, especially if you can't fit the entire score into memory at once. But it takes extra storage for the track numbers and may take extra work to manipulate the interleaved tracks.

The musical score format FORM SMUS is intended for simple loading and playback of small scores that fit entirely in main memory. So we chose to store its tracks separately.

There can be up to 255 tracks in a FORM SMUS. Each track is stored as a TRAK chunk. The count of tracks (the number of TRAK chunks) is recorded in the SHDR chunk at the beginning of the FORM SMUS. The TRAK chunks appear in numerical order 1, 2, 3, ... This is also priority order, most important track first. A player program that can handle up to N parallel tracks should read the first N tracks and ignore any others.

The different tracks in a score may have different lengths. This is true both of storage length and of playback duration.

Instrument Registers

Instrument reference. In SSSP, each note event points to a "timbre object" which supplies the "instrument" (the sound driver data) for that note. FORM SMUS stores these pointers as a "current instrument setting" for each track. It's just a run encoded version of the same information. SSSP uses a symbol table to hold all the pointers to "timbre object". SMUS uses INS1 chunks for the same purpose. They name the score's instruments.

The actual instrument data to use depends on the playback environment, but we want the score to be independent of environment. Different playback environments have different audio output hardware and different sound driver software. And there are channel allocation issues like how many output channels there are, which ones are polyphonic, and which I/O ports they're connected to. If you use MIDI to control the instruments, you get into issues of what kind of device is listening to each MIDI channel and what each of its presets sounds like. If you use computer-based instruments, you need driver- specific data like waveform tables and oscillator parameters.

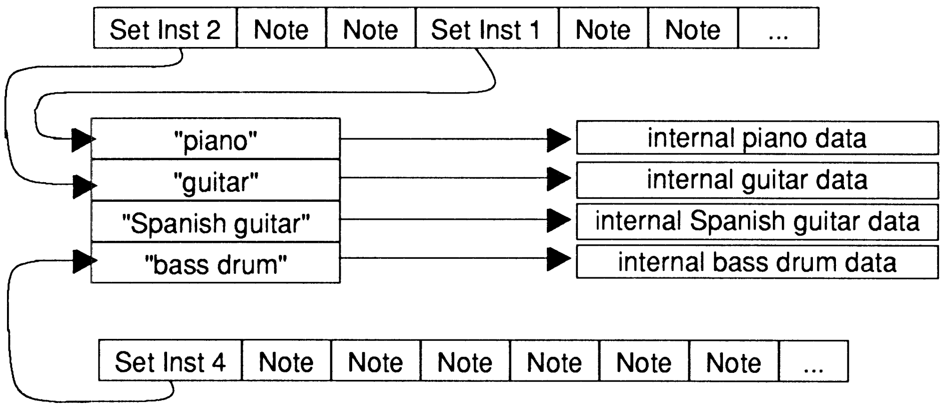
We just want some orchestration. If the score wants a "piano", we let the playback program find a "piano".

Instrument reference by name. A reference from a SMUS score to actual instrument data is normally by name. The score simply names the instrument, for instance "tubular bells". It's up to the player program to find suitable instrument data for its output devices. (More on locating instruments below.)

Instrument reference by MIDI channel and preset. A SMUS score can also ask for a specific MIDI channel number and preset number. MIDI programs may honor these specific requests. But these channel allocations can become obsolete or the score may be played without MIDI hardware. In such cases, the player program should fall back to instrument reference by name.

Instrument reference via instrument register. Each reference from a SMUS track to an instrument is via an "instrument register". Each track selects an instrument register which in turn points to the specific instrument data.

Each score has an array of instrument registers. Each track has a "current instrument setting", which is simply an index number into this array. This is like setting a raster image's pixel to a specific color number (a reference to a color value through a "color register") or setting a text character to a specific font number (a reference to a font through a "font register"). This is diagramed below:



Locating instrument data by name. "INS1" chunks in a SMUS score name the instruments to use for that score. The player program uses these names to locate instrument data.

To locate instrument data, the player performs these steps:

For each right register, check for a suitable instrument with the right name...
    {Suitable" means usable with an available output device and driver.}
    {Use case independent name comparisons.}

1. Initialize the instrument register to point to a built-in default instrument.
2. Every player program must have default instruments. Simple programs stop here. For fancier programs, the default instruments are a backstop in case the search fails.
3. Check any instrument FORMs embedded in the FORM SMUS. (This is an "instrument cache".)
4. Else check the default instruments.
5. Else search the local "instrument library". (The library might simply be a disk directory.)
6. If all else fails, display the desired instrument name and ask the user to pick an available one.

This algorithm can be implemented to varying degrees of fanciness. It's OK to stop searching after step 1, 2, 3, or 4. If exact instrument name matches fail, it's OK to try approximate matches. E.g., search for any kind of "guitar" if you can't find a "Spanish guitar". In any case, a player only has to search for instruments while loading a score.

When the embedded instruments are suitable, they save the program from asking the user to insert the "right" disk in a drive and searching that disk for the "right" instrument. But it's just a cache. In practice, we rarely move scores between environments so the cache often works. When the score is moved, embedded instruments must be discarded (a cache miss) and other instrument data used.

Be careful to distinguish an instrument's name from its filename—the contents name vs. container name. A musical instrument FORM should contain a NAME chunk that says what instrument it really is. Its filename, on the other hand, is a handle used to locate the FORM. Filenames are affected by external factors like drives, directories, and filename character and length limits. Instrument names are not.

Issue: Consider instrument naming conventions for consistency. Consider a naming convention that aids approximate matches. E.g., we could accept "guitar, bass1" if we didn't find "guitar, bass". Failing that, we could accept "guitar" or any name starting with "guitar".

Set instrument events. If the player implements the set-instrument score event, each track can change instrument numbers while playing. That is, it can switch between the loaded instruments.

Initial instrument settings. Each time a score is played, every track's running state information must be initialized. Specifically, each track's instrument number should be initialized to its track number. Track 1 to instrument 1, etc. It's as if each track began with a set-instrument event.

In this way, programs that don't implement the set-instrument event still assign an instrument to each track. The INS1 chunks imply these initial instrument settings.

MIDI Instruments

As mentioned above, A SMUS score can also ask for MIDI instruments. This is done by putting the MIDI channel and preset numbers in an INS1 chunk with the instrument name. Some programs will honor these requests while others will just find instruments by name.

MIDI Recorder and sequencer programs may simply transcribe the MIDI channel and preset commands in a recording session. For this purpose, set-MIDI-channel and set-MIDI-preset events can be embedded in a SMUS score's tracks. Most programs should ignore these events. An editor program that wants to exchange scores with such programs should recognize these events. It should let the user change them to the more general set-instrument events.

3. Standard Data and Property Chunks

A FORM SMUS contains a required property "SHDR" followed by any number of parallel "track" data chunks "TRAK". Optional property chunks such as "NAME", copyright "(c) ", and instrument reference "INS1" may also appear. Any of the properties may be shared over a LIST of FORMs SMUS by putting them in a PROP SMUS. [See the IFF reference.]

Required Property SHDR

The required property "SHDR" holds an SScoreHeader as defined in these C declarations and following documentation. An SHDR specifies global information for the score. It must appear before the TRAKs in a FORM SMUS.

```
#define ID_SMUS MakeID('S', 'M', 'U', 'S')
#define ID_SHDR MakeID('S', 'H', 'D', 'R')

typedef struct {
    UWORD tempo;    /* tempo, 128ths quarter note/minute */
    UBYTE volume;   /* overall playback volume 0 through 127 */
    UBYTE ctTrack;  /* count of tracks in the score */
    } SScoreHeader;
```

[Implementation details. In the C struct definitions in this memo, fields are filed in the order shown. A UBYTE field is packed into an 8-bit byte. Programs should set all "pad" fields to 0. MakeID is a C macro defined in the main IFF document and in the source file IFF.h.]

The field `tempo` gives the nominal tempo for all tracks in the score. It is expressed in 128ths of a quarter note per minute, i.e., 1 represents 1 quarter note per 128 minutes while 12800 represents 100 quarter notes per minute. You may think of this as a fixed point fraction with a 9-bit integer part and a 7-bit fractional part (to the right of the point). A coarse-tempoed program may simply shift `tempo` right by 7 bits to get a whole number of quarter notes per minute. The `tempo` field can store tempi in the range 0 up to 512. The playback program may adjust this tempo, perhaps under user control.

Actually, this global `tempo` could actually be just an initial tempo if there are any "set tempo" SEvents inside the score (see TRAK, below). Or the global tempo could be scaled by "scale tempo" SEvents inside the score. These are potential extensions that can safely be ignored by current programs. [See More SEvents To Be Defined, below.]

The field `volume` gives an overall nominal playback volume for all tracks in the score. The range of `volume` values 0 through 127 is like a MIDI key velocity value. The playback program may adjust this volume, perhaps under direction of a user "volume control".

Actually, this global volume level could be scaled by dynamic-mark SEvents inside the score (see TRAK, below).

The field `ctTrack` holds the count of tracks, i.e., the number of TRAK chunks in the FORM SMUS (see below). This information helps the reader prepare for the following data.

A playback program will typically load the score and call a driver routine `PlayScore(tracks, tempo, volume)`, supplying the `tempo` and `volume` from the SHDR chunk.

Optional Text Chunks NAME, (c), AUTH, ANNO

Several text chunks may be included in a FORM SMUS to keep ancillary information.

The optional property "NAME" names the musical score, for instance "Fugue in C".

The optional property "(c) " holds a copyright notice for the score. The chunk ID "(c) " serves the function of the copyright characters "© ". E.g., a "(c) " chunk containing "1986 Electronic Arts" means "© 1986 Electronic Arts".

The optional property "AUTH" holds the name of the score's author.

The chunk types "NAME", "(c) ", and "AUTH" are property chunks. Putting more than one NAME (or other) property in a FORM is redundant. Just the last NAME counts. A property should be shorter than 256 characters. Properties can appear in a PROP SMUS to share them over a LIST of FORMs SMUS.

The optional data chunk "ANNO" holds any text annotations typed in by the author.

An ANNO chunk is not a property chunk, so you can put more than one in a FORM SMUS. You can make ANNO chunks any length up to $2^{31} - 1$ characters, but 32767 is a practical limit. Since they're not properties, ANNO chunks don't belong in a PROP SMUS. That means they can't be shared over a LIST of FORMs SMUS.

Syntactically, each of these chunks contains an array of 8-bit ASCII characters in the range " " (SP, hex 20) through "~" (tilde, hex 7F), just like a standard "TEXT" chunk. [See "Strings, String Chunks, and String Properties" in "EA IFF 85" Electronic Arts Interchange File Format.] The chunk's `ckSize` field holds the count of characters.

```
#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the musical score's name.      */

#define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice.  */

#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the name of the score's author. */

#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations.     */
```

Remember to store a 0 pad byte after any odd-length chunk.

Optional Property INS1

The "INS1" chunks in a FORM SMUS identify the instruments to use for this score. A program can ignore INS1 chunks and stick with its built-in default instrument assignments. Or it can use them to locate instrument data. [See "Instrument Registers" in section 2, above.]

```
#define ID_INS1 MakeID('I', 'N', 'S', '1')

/* Values for the RefInstrument field "type".       */
#define INS1_Name  0        /* just use the name; ignore data1, data2  */
#define INS1_MIDI  1        /* <data1, data2> = MIDI <channel, preset> */

typedef struct {
    UBYTE register;     /* set this instrument register number */
    UBYTE type;         /* instrument reference type */
    UBYTE data1, data2; /* depends on the "type" field */
    CHAR  name[];       /* instrument name */
    } RefInstrument;
```

An INS1 chunk names the instrument for instrument register number `register`. The `register` field can range from 0 through 255. In practice, most scores will need only a few instrument registers.

The `name` field gives a text name for the instrument. The string length can be determined from the `ckSize` of the INS1 chunk. The string is simply an array of 8-bit ASCII characters in the range " " (SP, hex 20) through "~" (tilde, hex 7F).

Besides the instrument name, an INS1 chunk has two data numbers to help locate an instrument. The use of these data numbers is controlled by the `type` field. A value `type = INS1_Name` means just find an instrument by name. In this case, `data1` and `data2` should just be set to 0. A value `type = INS1_MIDI` means look for an instrument on MIDI channel # `data1`, preset # `data2`. Programs and computers without MIDI outputs will just ignore the MIDI data. They'll always look for the named instrument. Other values of the `type` field are reserved for future standardization.

See section 2, above, for the algorithm for locating instrument data by name.

Obsolete Property INST

The chunk type "INST" is obsolete in SMUS. It was revised to form the "INS1" chunk.

Data Chunk TRAK

The main contents of a score is stored in one or more TRAK chunks representing parallel "tracks". One TRAK chunk per track.

The contents of a TRAK chunk is an array of 16-bit "events" such as "note", "rest", and "set instrument". Events are really commands to a simple scheduler, stored in time order. The tracks can be polyphonic, that is, they can contain chorded "note" events.

Each event is stored as an "SEvent" record. ("SEvent" means "simple musical event".) Each SEvent has an 8-bit type field called an "sID" and 8 bits of type-dependent data. This is like a machine language instruction with an 8-bit opcode and an 8-bit operand.

This format is extensible since new event types can be defined in the future. The "note" and "rest" events are the only ones that every program must understand. *We will carefully design any new event types so that programs can safely skip over unrecognized events in a score.*

Caution: ID codes must be allocated by a central clearinghouse to avoid conflicts. Commodore Applications and Technical Support provides this clearinghouse service.

Here are the C type definitions for TRAK and SEvent and the currently defined sID values. Afterward are details on each SEvent.

```
#define ID_TRAK MakeID('T', 'R', 'A', 'K')
/* TRAK chunk contains an SEvent[].       */

/* SEvent: Simple musical event.          */
typedef struct {
    UBYTE sID;       /* SEvent type code   */
    UBYTE data;      /* sID-dependent data */
    } SEvent;

/* SEvent type codes "sID".                                      */
#define SID_FirstNote     0
#define SID_LastNote    127 /* sIDs in the range SID_FirstNote through
                             * SID_LastNote (sign bit = 0) are notes.
                             * The sID is the MIDI tone number (pitch).*/

#define SID_Rest        128 /* a rest (same data format as a note).   */
```

```
#define SID_Instrument  129 /* set instrument number for this track.    */
#define SID_TimeSig     130 /* set time signature for this track.       */
#define SID_KeySig      131 /* set key signature for this track.        */
#define SID_Dynamic     132 /* set volume for this track.               */
#define SID_MIDI_Chnl   133 /* set MIDI channel number (sequencers)     */
#define SID_MIDI_Preset 134 /* set MIDI preset number (sequencers)      */
#define SID_Clef        135 /* inline clef change.
                             * 0=Treble, 1=Bass, 2=Alto, 3=Tenor.(new) */
#define SID_Tempo       136 /* Inline tempo in beats per minute.(new)   */

/* SID values 144 through 159: reserved for Instant Music SEvents.      */

/* Remaining sID values up through 254: reserved for future
 * standardization.        */

#define SID_Mark        255 /* sID reserved for an end-mark in RAM.     */
```

Note and Rest SEvents

The note and rest SEvents `SID_FirstNote` through `SID_Rest` have the following structure overlaid onto the SEvent structure:

```
typedef struct {
    UBYTE    tone;          /* MIDI tone number 0 to 127; 128 = rest */
    unsigned chord    :1,   /* 1 = a chorded note                    */
             tieOut   :1,   /* 1 = tied to the next note or chord    */
             nTuplet  :2,   /* 0 = none, 1 = triplet, 2 = quintuplet,
                             * 3 = septuplet                         */
             dot      :1,   /* dotted note; multiply duration by 3/2 */
             division :3;   /* basic note duration is 2^-division: 0 =
                             * whole note, 1 = half note, 2 = quarter
                             * note, .... 7 = 128th note             */
    } SNote;
```

[Implementation details. Unsigned ":n" fields are packed into n bits in the order shown, most significant bit to least significant bit. An SNote fits into 16 bits like any other SEvent. Warning: Some compilers don't implement bit-packed fields properly. E.g., Lattice 68000 C pads a group of bit fields out to a LONG, which would make SNote take 5-bytes! In that situation, use the bit-field constants defined below.]

The SNote structure describes one "note" or "rest" in a track. The field `SNote.tone`, which is overlaid with the `SEvent.sID` field, indicates the MIDI tone number (pitch) in the range 0 through 127. A value of 128 indicates a rest.

The fields `nTuplet`, `dot`, and `division` together give the duration of the note or rest. The division gives the basic duration: whole note, half note, etc. The `dot` indicates if the note or rest is dotted. A dotted note is 3/2 as long as an undotted note. The value `nTuplet` (0 through 3) tells if this note or rest is part of an N-tuplet of order 1 (normal), 3, 5, or 7; an N-tuplet of order `(2 * nTuplet + 1)`. A triplet note is 2/3 as long as a normal note, while a quintuplet is 4/5 as long and a septuplet is 6/7 as long.

Putting these three fields together, the duration of the note or rest is

$$2^{-\text{division}} * \{1, 3/2\} * \{1, 2/3, 4/5, 6/7\}$$

These three fields are contiguous so you can easily convert to your local duration encoding by using the combined 6 bits as an index into a mapping table.

The field `chord` indicates if the note is chorded with the following note (which is supposed to have the same duration). A group of notes may be chorded together by setting the `chord` bit of all but the last one. (In the terminology of SSSP and GSCR, setting the `chord` bit to 1 makes the "entry delay" 0.) A monophonic-track player can simply ignore any SNote event whose `chord` bit is set, either by discarding it when reading the track or by skipping it when playing the track.

Programs that create polyphonic tracks are expected to store the most important note of each chord last, which is the note with the 0 `chord` bit. This way, monophonic programs will play the most important note of the chord. The most important note might be the chord's root note or its melody note.

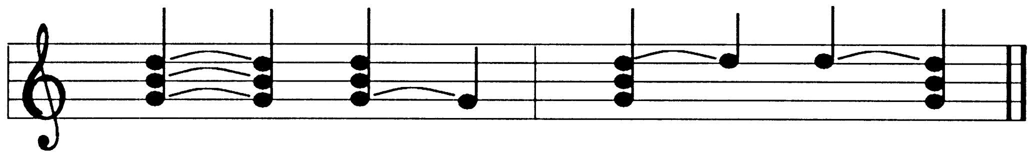
If the field `tieOut` is set, the note is tied to the following note in the track if the following note has the same pitch. A group of tied notes is played as a single note whose duration is the sum of the component durations. Actually, the tie mechanism ties a group of one or more chorded notes to another group of one or more chorded notes. Every note in a tied chord should have its `tieOut` bit set.

Of course, the `chord` and `tieOut` fields don't apply to `SID_Rest` SEvents.

Programs should be robust enough to ignore an unresolved tie, i.e., a note whose `tieOut` bit is set but isn't followed by a note of the same pitch. If that's true, monophonic-track programs can simply ignore chorded notes even in the presense of ties. That is, tied chords pose no extra problems.

The following diagram shows some combinations of notes and chords tied to notes and chords. The text below the staff has a column for each SNote SEvent to show the pitch, `chord` bit, and `tieOut` bit.

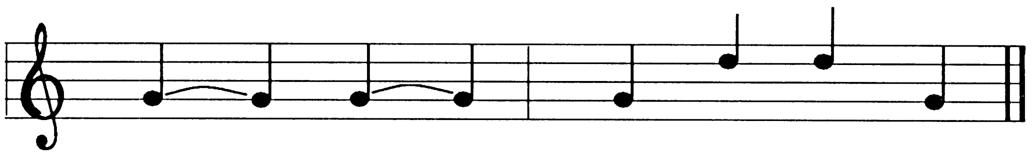
A treble staff with chords and ties:



Corresponding SNote values in the TRAK chunk:

```
Pitch:   D B G   D B G   D B G   G      D B G   B      B      D B G
chord:   c c -   c c -   c c -   -      c c -   -      -      c c -
tieOut:  t t t   - - -   t t t   -      t t t   -      t      - - -
```

If you read the above track into a monophonic-track program, it'll strip out the chorded notes and ignore unresolved ties. You'll end up with:



```
Pitch:  G        G        G        G        G        B        B        G
chord:  -        -        -        -        -        -        -        -
tieOut: t        -        t        -       (t)       -       (t)       -
```

A rest event `(sID = SID_Rest)` has the same `SEvent.data` field as a note. It tells the duration of the rest. The `chord` and `tieOut` fields of rest events are ignored.

Within a TRAK chunk, note and rest events appear in time order.

Instead of the bit-packed structure SNote, it might be easier to assemble `data` values by or-ing constants and to disassemble them by masking and shifting. In that case, use the following definitions.

```
#define noteChord  (1<<7)          /* note is chorded to next note    */
#define noteTieOut (1<<6)          /* tied to next note/chord         */

#define noteNShift 4               /* shift count for nTuplet field  */
#define noteN3     (1<<noteNShift) /* note is a triplet               */
#define noteN5     (2<<noteNShift) /* note is a quintuplet            */
#define noteN7     (3<<noteNShift) /* note is a septuplet             */
#define noteNMask  noteN7          /* bit mask for the nTuplet field */

#define noteDot    (1<<3)          /* note is dotted                  */

#define noteD1     0              /* whole note division             */
#define noteD2     1              /* half note division              */
#define noteD4     2              /* quarter note division           */
#define noteD8     3              /* eighth note division            */
#define noteD16    4              /* sixteenth note division         */
#define noteD32    5              /* thirty-secondth note division   */
#define noteD64    6              /* sixty-fourth note division      */
#define noteD128   7              /* 1/128 note division             */
#define noteDMask  noteD128       /* bit mask for the division field */

#define noteDurMask 0x3F          /* mask for combined duration fields*/
```

Note: The remaining SEvent types are optional. A writer program doesn't have to generate them. A reader program can safely ignore them.

Set Instrument SEvent

One of the running state variables of every track is an instrument number. An instrument number is the array index of an "instrument register", which in turn points to an instrument. (See "Instrument Registers", in section 2.) This is like a color number in a bitmap; a reference to a color through a "color register".

The initial setting for each track's instrument number is the track number. Track 1 is set to instrument 1, etc. Each time the score is played, every track's instrument number should be reset to the track number.

The SEvent `SID_Instrument` changes the instrument number for a track, that is, which instrument plays the following notes. Its `SEvent.data` field is an instrument register number in the range 0 through 255. If a program doesn't implement the `SID_Instrument` event, each track is fixed to one instrument.

Set Time Signature SEvent

The SEvent `SID_TimeSig` sets the time signature for the track. A "time signature" SEvent has the following structure overlaid on the SEvent structure:

```
typedef struct {
    UBYTE    type;          /* = SID_TimeSig */
    unsigned timeNSig :5,   /* time sig. "numerator" is timeNSig + 1 */
             timeDSig :3;   /* time sig. "denominator" is 2^timeDSig:*
                             * 0 = whole note, 1 = half note, 2 =    *
                             * quarter note,....7 = 128th note       */
    } STimeSig;
```

[Implementation details. Unsigned ":n" fields are packed into n bits in the order shown, most significant bit to least significant bit. An STimeSig fits into 16 bits like any other SEvent. Warning: Some compilers don't implement bit-packed fields properly. E.g., Lattice C pads a group of bit fields out to a LONG, which would make an STimeSig take 5-bytes! In that situation, use the bit-field constants defined below.]

The field type contains the value `SID_TimeSig`, indicating that this SEvent is a "time signature" event. The field `timeNSig` indicates the time signature "numerator" is `timeNSig + 1`, that is, 1 through 32 beats per measure. The field `timeDSig` indicates the time signature "denominator" is $2^{timeDSig}$, that is each "beat" is a $2^{-timeDSig}$ note (see SNote `division`, above). So 4/4 time is expressed as `timeNSig = 3`, `timeDSig = 2`.

The default time signature is 4/4 time. Be aware that the time signature has no effect on the score's playback. Tempo is uniformly expressed in quarter notes per minute, independent of time signature. (Quarter notes per minute would equal beats per minute only if `timeDSig = 2`, n/4 time). Nonetheless, any program that has time signatures should put them at the beginning of each TRAK when creating a FORM SMUS because music editors need them.

Instead of the bit-packed structure STimeSig, it might be easier to assemble `data` values by or-ing constants and to disassemble them by masking and shifting. In that case, use the following definitions.

```
#define timeNMask  0xF8 /* bit mask for the timeNSig field      */
#define timeNShift 3    /* shift count for  timeNSig field      */
#define timeDMask  0x07 /* bit mask for the timeDSig field      */
```

Key Signature SEvent

An SEvent `SID_KeySig` sets the key signature for the track. Its `data` field is a UBYTE number encoding a major key:

data	key	music notation	data	key	music notation
0	C maj				
1	G	#	8	F	b
2	D	##	9	Bb	bb
3	A	###	10	Eb	bbb
4	E	####	11	Ab	bbbb
5	B	#####	12	Db	bbbbb
6	F#	######	13	Gb	bbbbbb
7	C#	#######	14	Cb	bbbbbbb

A `SID_KeySig` SEvent changes the key for the following notes in that track. C major is the default key in every track before the first `SID_KeySig` SEvent.

Dynamic Mark SEvent

An SEvent `SID_Dynamic` represents a dynamic mark like *ppp* and *fff* in Common Music Notation. Its `data` field is a MIDI key velocity number 0 through 127. This sets a "volume control" for following notes in the track. This "track volume control" is scaled by the overall score `volume` in the SHDR chunk. The default dynamic level is 127 (full volume).

Set MIDI Channel SEvent

The SEvent `SID_MIDI_Chnl` is for recorder programs to record the set-MIDI-channel low level event. The `data` byte contains a MIDI channel number. Other programs should use instrument registers instead.

Set MIDI Preset SEvent

The SEvent `SID_MIDI_Preset` is for recorder programs to record the set-MIDI-preset low level event. The `data` byte contains a MIDI preset number. Other programs should use instrument registers instead.

Instant Music Private SEvents

Sixteen SEvents are used for private data for the Instant Music program. SID values 144 through 159 are reserved for this purpose. Other programs should skip over these SEvents.

End-Mark SEvent

The SEvent type `SID_Mark` is reserved for an end marker in working memory. *This event is never stored in a file.* It may be useful if you decide to use the filed TRAK format intact in working memory.

More SEvents To Be Defined

More SEvents can be defined in the future. The sID codes 133 through 143 and 160 through 254 are reserved for future needs. Caution: sID codes must be allocated by a central "clearinghouse" to avoid conflicts.

The following SEvent types are under consideration and should not yet be used.

Issue: A "change tempo" SEvent changes tempo during a score. Changing the tempo affects all tracks, not just the track containing the change tempo event.

One possibility is a "scale tempo" SEvent `SID_ScaleTempo` that rescales the global tempo:

```
        currentTempo := globalTempo * (data + 1) / 128
```

This can scale the global tempo (in the SHDR) anywhere from x1/128 to x2 in roughly 1% increments.

An alternative is two events `SID_SetHTempo` and `SID_SetLTempo`. `SID_SetHTempo` gives the high byte and `SID_SetLTempo` gives the low byte of a new tempo setting, in 128ths quarter note/minute. `SetHTempo` automatically sets the low byte to 0, so the `SetLTempo` event isn't needed for coarse settings. In this scheme, the SHDR's `tempo` is simply a starting tempo.

An advantage of `SID_ScaleTempo` is that the playback program can just alter the global tempo to adjust the overall performance time and still easily implement tempo variations during the score. But the "set tempo" SEvent may be simpler to generate.

Issue: The events `SID_BeginRepeat` and `SID_EndRepeat` define a repeat span for one track. The span of events between a `BeginRepeat` and an `EndRepeat` is played twice. The `SEvent.data` field in the `BeginRepeat` event could give an iteration count, 1 through 255 times or 0 for "repeat forever".

Repeat spans can be nested. All repeat spans automatically end at the end of the track.

An event `SID_Ending` begins a section like "first ending" or "second ending". The SEvent.data field gives the ending number. This `SID_Ending` event only applies to the innermost repeat group. (Consider generalizing it.)

A more general alternative is a "subtrack" or "subscore" event. A "subtrack" event is essentially a "subroutine call" to another series of SEvents. This is a nice way to encode all the possible variations of repeats, first endings, codas, and such.

To define a subtrack, we must demark its start and end. One possibility is to define a relative branch-to-subtrack event `SID_BSR` and a return-from-subtrack event `SID_RTS`. The 8-bit `data` field in the `SID_BSR` event can reach as far as 512 SEvents. A second possibility is to call a subtrack by index number, with an IFF chunk outside the TRAK defining the start and end of all subtracks. This is very general since a portion of one subtrack can be used as another subtrack. It also models the tape recording practice of first "laying down a track" and then selecting portions of it to play and repeat. To embody the music theory idea of playing a sequence like "ABBA", just compose the "main" track entirely of subtrack events. A third possibility is to use a numbered subtrack chunk "STRK" for each subroutine.

4. Private Chunks

As in any IFF FORM, there can be private chunks in a FORM SMUS that are designed for one particular program to store its private information. All IFF reader programs skip over unrecognized chunks, so the presense of private chunks can't hurt.

Instant Music stores some global score information in a chunk of ID "IRev" and some other information in a chunk of ID "BIAS".

Appendix A. Quick Reference

Type Definitions

Here's a collection of the C type definitions in this memo. In the "struct" type definitions, fields are filed in the order shown. A UBYTE field is packed into an 8-bit byte. Programs should set all "pad" fields to 0.

```
#define ID_SMUS MakeID('S', 'M', 'U', 'S')
#define ID_SHDR MakeID('S', 'H', 'D', 'R')

typedef struct {
    UWORD tempo;    /* tempo, 128ths quarter note/minute           */
    UBYTE volume;   /* overall playback volume 0 through 127       */
    UBYTE ctTrack;  /* count of tracks in the score                */
    } SScoreHeader;

#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the musical score's name.         */

#define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice.    */

#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the name of the score's author.   */

#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations.        */

#define ID_INS1 MakeID('I', 'N', 'S', '1')
/* Values for the RefInstrument field "type".                      */

#define INS1_Name 0 /* just use the name; ignore data1, data2      */
#define INS1_MIDI 1 /* <data1, data2> = MIDI <channel, preset>       */

typedef struct {
    UBYTE register;     /* set this instrument register number     */
    UBYTE type;         /* instrument reference type               */
    UBYTE data1, data2; /* depends on the "type" field             */
    CHAR  name[];       /* instrument name                         */
    } RefInstrument;

#define ID_TRAK MakeID('T', 'R', 'A', 'K')
/* TRAK chunk contains an SEvent[].                                */

/* SEvent: Simple musical event.                                   */
typedef struct {
    UBYTE sID;      /* SEvent type code                            */
    UBYTE data;     /* sID-dependent data                          */
    } SEvent;


/* SEvent type codes "sID".                                        */
#define SID_FirstNote     0
#define SID_LastNote    127 /* sIDs in the range SID_FirstNote through
                             * SID_LastNote (sign bit = 0) are notes. The
                             * sID is the MIDI tone number (pitch). */
#define SID_Rest        128 /* a rest (same data format as a note). */

#define SID_Instrument  129 /* set instrument number for this track.*/
#define SID_TimeSig     130 /* set time signature for this track.   */
#define SID_KeySig      131 /* set key signature for this track.    */
#define SID_Dynamic     132 /* set volume for this track.           */
#define SID_MIDI_Chnl   133 /* set MIDI channel number (sequencers) */
#define SID_MIDI_Preset 134 /* set MIDI preset number (sequencers)  */
#define SID_Clef        135 /* inline clef change.                  *
                             * 0=Treble, 1=Bass, 2=Alto, 3=Tenor.   */
#define SID_Tempo       136 /* Inline tempo in beats per minute.    */

/* SID values 144 through 159: reserved for Instant Music SEvents.  */

/* Remaining sID values up through 254: reserved for future
 * standardization.                                                 */
```

```
#define SID_Mark        255 /* sID reserved for an end-mark in RAM. */

/* SID_FirstNote..SID_LastNote, SID_Rest SEvents                   */

typedef struct {
    UBYTE    tone;          /* MIDI tone number 0 to 127; 128 = rest */
    unsigned chord    :1,   /* 1 = a chorded note                    */
             tieOut   :1,   /* 1 = tied to the next note or chord    */
             nTuplet  :2,   /* 0 = none, 1 = triplet, 2 = quintuplet,
                             * 3 = septuplet                         */
             dot      :1,   /* dotted note; multiply duration by 3/2 */
             division :3;   /* basic note duration is 2-division: 0 = whole
                             * note, 1 = half note, 2 = quarter note,
                             * 7 = 128th note                        */
            } SNote;

#define noteChord  (1<<7)   /* note is chorded to next note          */

#define noteTieOut (1<<6)   /* tied to next note/chord               */

#define noteNShift 4        /* shift count for nTuplet field         */

#define noteN3     (1<<noteNShift)  /* note is a triplet             */
#define noteN5     (2<<noteNShift)  /* note is a quintuplet          */
#define noteN7     (3<<noteNShift)  /* note is a septuplet           */

#define noteNMask  noteN7   /* bit mask for the nTuplet field        */

#define noteDot    (1<<3)   /* note is dotted                        */


#define noteD1     0        /* whole note division                   */
#define noteD2     1        /* half note division                    */
#define noteD4     2        /* quarter note division                 */
#define noteD8     3        /* eighth note division                  */
#define noteD16    4        /* sixteenth note division               */
#define noteD32    5        /* thirty-secondth note division         */
#define noteD64    6        /* sixty-fourth note division            */
#define noteD128   7        /* 1/128 note division                   */
#define noteDMask  noteD128 /* bit mask for the division field       */


#define noteDurMask 0x3F    /* mask for combined duration fields     */


/* SID_Instrument SEvent                                             */
/* "data" value is an instrument register number 0 through 255.      */

/* SID_TimeSig SEvent                                                */
typedef struct {
    UBYTE    type;          /* = SID_TimeSig                         */
    unsigned timeNSig :5,   /* time sig. "numerator" is timeNSig + 1 */
             timeDSig :3;   /* time sig. "denominator" is 2^timeDSig: *
                             * 0 = whole note, 1 = half note, 2 =    *
                             * quarter note,.... 7 = 128th note      */
    } STimeSig;

#define timeNMask  0xF8     /* bit mask for the timeNSig field       */
#define timeNShift 3        /* shift count for  timeNSig field       */

#define timeDMask  0x07     /* bit mask for the timeDSig field       */

/* SID_KeySig SEvent                                                 */
/* "data" value 0 = Cmaj; 1 through 7 = G,D,A,E,B,F#,C#;             *
 * 8 through 14 = F,Bb,Eb,Ab,Db,Gb,Cb.                               */

/* SID_Dynamic SEvent                                                */
/* "data" value is a MIDI key velocity 0..127.                       */
```

SMUS Regular Expression

Here's a regular expression summary of the FORM SMUS syntax. This could be an IFF file or part of one.

```
SMUS      ::= "FORM" #{ "SMUS" SHDR [NAME] [Copyright] [AUTH] [IRev]
                        ANNO* INS1*  TRAK*  InstrForm* }

SHDR      ::= "SHDR" #{ SScoreHeader    }
NAME      ::= "NAME" #{ CHAR*    } [0]
Copyright ::= "(c) " #{ CHAR*    } [0]
AUTH      ::= "AUTH" #{ CHAR*    } [0]
IRev      ::= "IRev" #{ ...      }

ANNO      ::= "ANNO" #{ CHAR*    } [0]
INS1      ::= "INS1" #{ RefInstrument    } [0]

TRAK      ::= "TRAK" #{ SEvent* }
InstrForm ::= "FORM" #{ ...      }
```

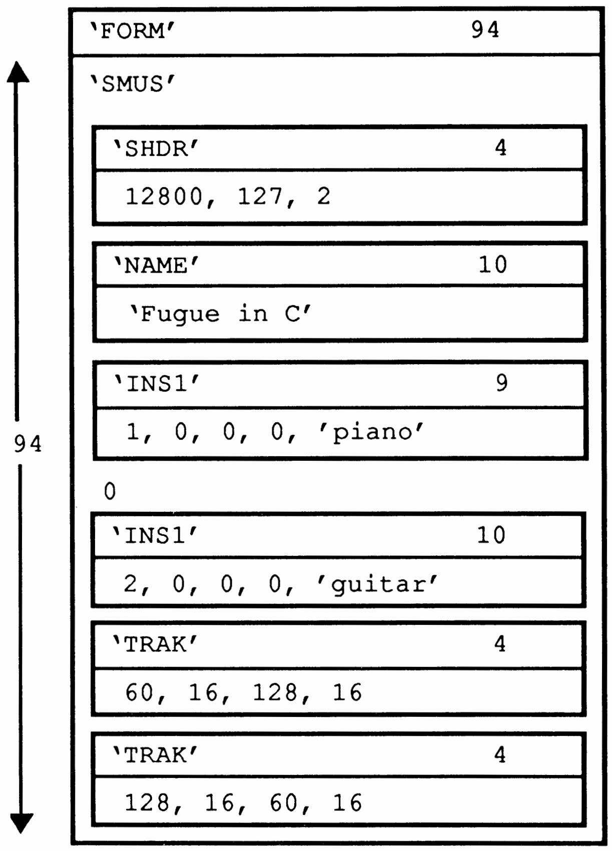
The token "#" represents a `ckSize` LONG count of the following {braced} data bytes. Literal items are shown in "quotes", [square bracket items] are optional, and "*" means 0 or more replications. A sometimes-needed pad byte is shown as "`[0]`".

Actually, the order of chunks in a FORM SMUS is not as strict as this regular expression indicates. The SHDR, NAME, Copyright, AUTH, IRev, ANNO, and INS1 chunks may appear in any order, as long as they precede the TRAK chunks.

The chunk "`InstrForm`" represents any kind of instrument data FORM embedded in the FORM SMUS. For example, see the document "8SVX" IFF 8-Bit Sampled Voice. Of course, a recipient program will ignore an instrument FORM if it doesn't recognize that FORM type.

Appendix B. SMUS Example

Here's a box diagram for a simple example, a SMUS with two instruments and two tracks. Each track contains 1 note event and 1 rest event.



The "0" after the first INS1 chunk is a pad byte.

Appendix C. Standards Committee

The following people contributed to the design of this SMUS standard:

Ralph Bellafatto, Cherry Lane Technologies
Geoff Brown, Uhuru Sound Software
Steve Hayes, Electronic Arts
Jerry Morrison, Electronic Arts

"8SVX" IFF 8-Bit Sampled Voice

Date: February 7, 1985 (Re-Typeset Oct, 1988 Commodore-Amiga, Inc.)
From: Steve Hayes and Jerry Morrison, Electronic Arts
Status: Adopted

1. Introduction

This is the IFF supplement for FORM "8SVX". An 8SVX is an IFF "data section" or "FORM" (which can be an IFF file or a part of one) containing a digitally sampled audio voice consisting of 8-bit samples. A voice can be a one-shot sound or—with repetition and pitch scaling—a musical instrument. "EA IFF 85" is Electronic Arts' standard interchange file format. [See "EA IFF 85" Standard for Interchange Format Files.]

The 8SVX format is designed for playback hardware that uses 8-bit samples attenuated by a volume control for good overall signal-to-noise ratio. So a FORM 8SVX stores 8-bit samples and a volume level.

A similar data format (or two) will be needed for higher resolution samples (typically 12 or 16 bits). Properly converting a high resolution sample down to 8 bits requires one pass over the data to find the minimum and maximum values and a second pass to scale each sample into the range -128 through 127. So it's reasonable to store higher resolution data in a different FORM type and convert between them.

For instruments, FORM 8SVX can record a repeating waveform optionally preceded by a startup transient waveform. These two recorded signals can be pre-synthesized or sampled from an acoustic instrument. For many instruments, this representation is compact. FORM 8SVX is less practical for an instrument whose waveform changes from cycle to cycle like a plucked string, where a long sample is needed for accurate results.

FORM 8SVX can store an "envelope" or "amplitude contour" to enrich musical notes. A future voice FORM could also store amplitude, frequency, and filter modulations.

FORM 8SVX is geared for relatively simple musical voices, where one waveform per octave is sufficient, the waveforms for the different octaves follow a factor-of-two size rule, and one envelope is adequate for all octaves. You could store a more general voice as a LIST containing one or more FORMs 8SVX per octave. A future voice FORM could go beyond one "one-shot" waveform and one "repeat" waveform per octave.

Section 2 defines the required property sound header "VHDR", optional properties name "NAME", copyright "(c) ", and author "AUTH", the optional annotation data chunk "ANNO", the required data chunk "BODY", and optional envelope chunks "ATAK" and "RLSE". These are the "standard" chunks. Specialized chunks for private or future needs can be added later, e.g., to hold a frequency contour or Fourier series coefficients. The 8SVX syntax is summarized in Appendix A as a regular expression and in Appendix B as an example box diagram. Appendix C explains the optional Fibonacci-delta compression algorithm.

Reference:

"EA IFF 85" Standard for Interchange Format Files describes the conventions for all IFF files.
Amiga® is a registered trademark of Commodore-Amiga, Inc.
Electronic Arts™ is a trademark of Electronic Arts.

2. Standard Data and Property Chunks

FORM 8SVX stores all the waveform data in one body chunk "BODY". It stores playback parameters in the required header chunk "VHDR". "VHDR" and any optional property chunks "NAME", "(c) ", and "AUTH" must all appear before the BODY chunk. Any of these properties may be shared over a LIST of FORMs 8SVX by putting them in a PROP 8SVX. [See "EA IFF 85" Standard for Interchange Format Files.]

Background

There are two ways to use FORM 8SVX: as a one-shot sampled sound or as a sampled musical instrument that plays "notes". Storing both kinds of sounds in the same kind of FORM makes it easy to play a one-shot sound as an instrument or an instrument as a one-note sound.

A one-shot sound is a series of audio data samples with a nominal playback rate and amplitude. The recipient program can optionally adjust or modulate the amplitude and playback data rate.

For musical instruments, the idea is to store a sampled (or pre-synthesized) waveform that will be parameterized by pitch, duration, and amplitude to play each "note". The creator of the FORM 8SVX can supply a waveform per octave over a range of octaves for this purpose. The intent is to perform a pitch by selecting the closest octave's waveform and scaling the playback data rate. An optional "one-shot" waveform supplies an arbitrary startup transient, then a "repeat" waveform is iterated as long as necessary to sustain the note.

A FORM 8SVX can also store an envelope to modulate the waveform. Envelopes are mostly useful for variable-duration notes but could be used for one-shot sounds, too.

The FORM 8SVX standard has some restrictions. For example, each octave of data must be twice as long as the next higher octave. Most sound driver software and hardware imposes additional restrictions. E.g., the Amiga sound hardware requires an even number of samples in each one-shot and repeat waveform.

Required Property VHDR

The required property "VHDR" holds a Voice8Header structure as defined in these C declarations and following documentation. This structure holds the playback parameters for the sampled waveforms in the BODY chunk. (See "Data Chunk BODY", below, for the storage layout of these waveforms.)

```
#define ID_8SVX MakeID('8', 'S', 'V', 'X')
#define ID_VHDR MakeID('V', 'H', 'D', 'R')

typedef LONG Fixed;    /* A fixed-point value, 16 bits to the left of the
                          point and 16 to the right.  A Fixed is a number
                          of 2^16ths, i.e., 65536ths.                    */
#define Unity 0x10000L /* Unity = Fixed 1.0 = maximum volume             */

/* sCompression: Choice of compression algorithm applied to the samples. */
#define sCmpNone       0    /* not compressed                            */
#define sCmpFibDelta   1    /* Fibonacci-delta encoding (Appendix C)     */
                            /* Can be more kinds in the future.          */
typedef struct {
    ULONG oneShotHiSamples, /* # samples in the high octave 1-shot part */
          repeatHiSamples,  /* # samples in the high octave repeat part */
          samplesPerHiCycle;/* # samples/cycle in high octave, else 0   */
    UWORD samplesPerSec;    /* data sampling rate                       */
    UBYTE ctOctave,         /* # octaves of waveforms                   */
          sCompression;     /* data compression technique used          */
    Fixed volume;           /* playback volume from 0 to Unity (full
                             * volume). Map this value into the output
                             * hardware's dynamic range.                */
    } Voice8Header;
```

[Implementation details. Fields are filed in the order shown. The UBYTE fields are byte-packed (2 per 16-bit word). MakeID is a C macro defined in the main IFF document and in the source file IFF.h.]

A FORM 8SVX holds waveform data for one or more octaves, each containing a one-shot part and a repeat part. The fields `oneShotHiSamples` and `repeatHiSamples` tell the number of audio samples in the two parts of the highest frequency octave. Each successive (lower frequency) octave contains twice as many data samples in both its one-shot and repeat parts. One of these two parts can be empty across all octaves.

Note: Most audio output hardware and software has limitations. For example the Amiga computer has sound hardware that requires that all one-shot and repeat parts have even numbers of samples. Amiga sound driver software should adjust an odd-sized waveform, ignore an odd-sized lowest octave, or ignore odd 8SVX FORMs altogether. Some other output devices require all sample sizes to be powers of two.

The field `samplesPerHiCycle` tells the number of samples/cycle in the highest frequency octave of data, or else 0 for "unknown". Each successive (lower frequency) octave contains twice as many samples/cycle. The `samplesPerHiCycle` value is needed to compute the data rate for a desired playback pitch.

Actually, `samplesPerHiCycle` is an average number of samples/cycle. If the one-shot part contains pitch bends, store the samples/cycle of the repeat part in `samplesPerHiCycle`. The division `repeatHiSamples/samplesPerHiCycle` should yield an integer number of cycles. (When the repeat waveform is repeated, a partial cycle would come out as a higher-frequency cycle with a "click".)

More limitations: some Amiga music drivers require `samplesPerHiCycle` to be a power of two in order to play the FORM 8SVX as a musical instrument in tune. They may even assume `samplesPerHiCycle` is a particular power of two without checking. (If `samplesPerHiCycle` is different by a factor of two, the instrument will just be played an octave too low or high.)

The field `samplesPerSec` gives the sound sampling rate. A program may adjust this to achieve frequency shifts or vary it dynamically to achieve pitch bends and vibrato. A program that plays a FORM 8SVX as a musical instrument would ignore `samplesPerSec` and select a playback rate for each musical pitch.

The field `ctOctave` tells how many octaves of data are stored in the BODY chunk. See "Data Chunk BODY", below, for the layout of the octaves.

The field `sCompression` indicates the compression scheme, if any, that was applied to the entire set of data samples stored in the BODY chunk. This field should contain one of the values defined above. Of course, the matching decompression algorithm must be applied to the BODY data before the sound can be played. (The Fibonacci-delta encoding scheme `sCmpFibDelta` is described in Appendix C.) Note that the whole series of data samples is compressed as a unit.

The field `volume` gives an overall playback volume for the waveforms (all octaves). It lets the 8-bit data samples use the full range -128 through 127 for good signal-to-noise ratio. The playback program should multiply this value by a "volume control" and perhaps by a playback envelope (see ATAK and RLSE, below).

Recording a one-shot sound. To store a one-shot sound in a FORM 8SVX, set `oneShotHiSamples` = number of samples, `repeatHiSamples` = 0 , `samplesPerHiCycle` = 0, `samplesPerSec` = sampling rate, and `ctOctave` = 1. Scale the signal amplitude to the full sampling range -128 through 127. Set `volume` so the sound will playback at the desired volume level. If you set the `samplesPerHiCycle` field properly, the data can also be used as a musical instrument.

Experiment with data compression. If the decompressed signal sounds OK, store the compressed data in the BODY chunk and set `sCompression` to the compression code number.

Recording a musical instrument. To store a musical instrument in a FORM 8SVX, first record or synthesize as many octaves of data as you want to make available for playback. Set `ctOctave` to the count of octaves. From the recorded data, excerpt an integral number of steady state cycles for the repeat part and set `repeatHiSamples` and `samplesPerHiCycle`. Either excerpt a startup transient waveform and set `oneShotHiSamples`, or else set `oneShotHiSamples` to 0. Remember, the one-shot and repeat parts of each octave must be twice as long as those of the next higher octave. Scale the signal amplitude to the full sampling range and set volume to adjust the instrument playback volume. If you set the `samplesPerSec` field properly, the data can also be used as a one-shot sound.

A distortion-introducing compressor like `sCmpFibDelta` is not recommended for musical instruments, but you might try it anyway.

Typically, creators of FORM 8SVX record an acoustic instrument at just one frequency. Decimate (down-sample with filtering) to compute higher octaves. Interpolate to compute lower octaves.

If you sample an acoustic instrument at different octaves, you may find it hard to make the one-shot and repeat waveforms follow the factor-of-two rule for octaves. To compensate, lengthen an octave's one-shot part by appending replications of the repeating cycle or prepending zeros. (This will have minimal impact on the sound's start time.) You may be able to equalize the ratio of one-shot-samples to repeat-samples across all octaves.

Note that a "one-shot sound" may be played as a "musical instrument" and vice-versa. However, an instrument player depends on `samplesPerHiCycle`, and a one-shot player depends on `samplesPerSec`.

Playing a one-shot sound. To play any FORM 8SVX data as a one-shot sound, first select an octave if `ctOctave` > 1. (The lowest-frequency octave has the greatest resolution.) Play the one-shot samples then the repeat samples, scaled by volume, at a data rate of `samplesPerSec`. Of course, you may adjust the playback rate and volume. You can play out an envelope, too. (See ATAK and RLSE, below.)

Playing a musical note. To play a musical note using any FORM 8SVX, first select the nearest octave of data from those available. Play the one-shot waveform then cycle on the repeat waveform as long as needed to sustain the note. Scale the signal by `volume`, perhaps also by an envelope, and by a desired note volume. Select a playback data rate s samples/second to achieve the desired frequency (in Hz):

```
        frequency = s / samplesPerHiCycle
```

for the highest frequency octave.

The idea is to select an octave and one of 12 sampling rates (assuming a 12-tone scale). If the FORM 8SVX doesn't have the right octave, you can decimate or interpolate from the available data.

When it comes to musical instruments, FORM 8SVX is geared for a simple sound driver. Such a driver uses a single table of 12 data rates to reach all notes in all octaves. That's why 8SVX requires each octave of data to have twice as many samples as the next higher octave. If you restrict `samplesPerHiCycle` to a power of two, you can use a predetermined table of data rates.

Optional Text Chunks NAME, (c), AUTH, ANNO

Several text chunks may be included in a FORM 8SVX to keep ancillary information.

The optional property "NAME" names the voice, for instance "tubular bells".

The optional property "(c) " holds a copyright notice for the voice. The chunk ID "(c) " serves as the copyright characters "©". E.g., a "(c) " chunk containing "1986 Electronic Arts" means "© 1986 Electronic Arts".

The optional property "AUTH" holds the name of the instrument's "author" or "creator".

The chunk types "NAME", "(c) ", and "AUTH" are property chunks. Putting more than one NAME (or other) property in a FORM is redundant. Just the last NAME counts. A property should be shorter than 256 characters. Properties can appear in a PROP 8SVX to share them over a LIST of FORMs 8SVX.

The optional data chunk "ANNO" holds any text annotations typed in by the author.

An ANNO chunk is not a property chunk, so you can put more than one in a FORM 8SVX. You can make ANNO chunks any length up to 2^{31} - 1 characters, but 32767 is a practical limit. Since they're not properties, ANNO chunks don't belong in a PROP 8SVX. That means they can't be shared over a LIST of FORMs 8SVX.

Syntactically, each of these chunks contains an array of 8-bit ASCII characters in the range " " (SP, hex 20) through "~" (tilde, hex 7F), just like a standard "TEXT" chunk. [See "Strings, String Chunks, and String Properties" in "EA IFF 85" Electronic Arts Interchange File Format.] The chunk's `ckSize` field holds the count of characters.

```
#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the voice's name.          */

define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice.*/

#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the author's name.         */

#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations. */
```

Remember to store a 0 pad byte after any odd-length chunk.

Optional Data Chunks ATAK and RLSE

The optional data chunks ATAK and RLSE together give a piecewise-linear "envelope" or "amplitude contour". This contour may be used to modulate the sound during playback. It's especially useful for playing musical notes of variable durations. Playback programs may ignore the supplied envelope or substitute another.

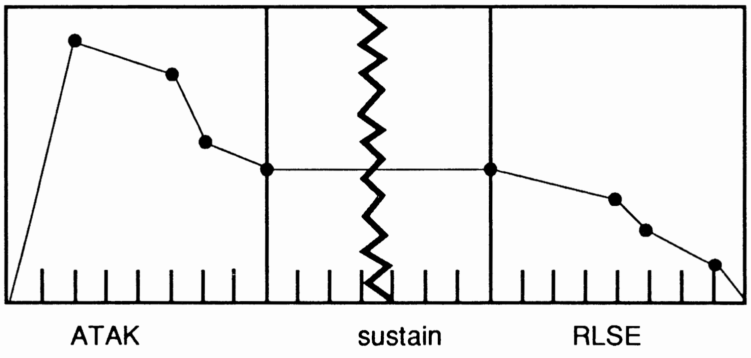
```
#define ID_ATAK MakeID('A', 'T', 'A', 'K')
#define ID_RLSE MakeID('R', 'L', 'S', 'E')

typedef struct {
    UWORD duration; /* segment duration in milliseconds, > 0 */
    Fixed dest;     /* destination volume factor             */
    } EGPoint;

/* ATAK and RLSE chunks contain an EGPoint[], piecewise-linear envelope.*/
/* The envelope defines a function of time returning Fixed values. It's *
 * used to scale the nominal volume specified in the Voice8Header.      */
```

To explain the meaning of the ATAK and RLSE chunks, we'll overview the envelope generation algorithm. Start at 0 volume, step through the ATAK contour, then hold at the sustain level (the last ATAK EGPoint's `dest`), and then step through the RLSE contour. Begin the release at the desired note stop time minus the total duration of the release contour (the sum of the RLSE EGPoints' `durations`). The attack contour should be cut short if the note is shorter than the release contour.

The envelope is a piecewise-linear function. The envelope generator interpolates between the EGPoints.

Remember to multiply the envelope function by the nominal voice header `volume` and by any desired note volume.

Figure 1 shows an example envelope. The attack period is described by 4 EGPoints in an ATAK chunk. The release period is described by 4 EGPoints in a RLSE chunk. The sustain period in the middle just holds the final ATAK level until it's time for the release.



Note: The number of EGPoints in an ATAK or RLSE chunk is its `ckSize / sizeof(EGPoint)`. In RAM, the playback program may terminate the array with a 0 `duration` EGPoint.

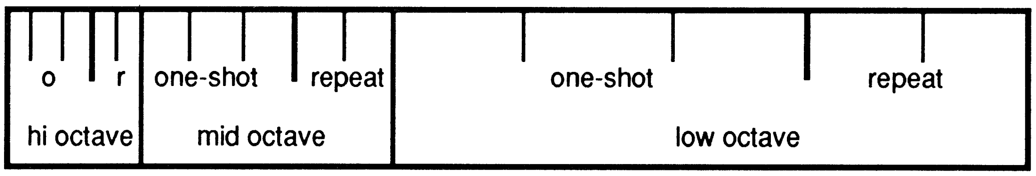
Issue: Synthesizers also provide frequency contour (pitch bend), filtering contour (wah-wah), amplitude oscillation (tremolo), frequency oscillation (vibrato), and filtering oscillation (leslie). In the future, we may define optional chunks to encode these modulations. The contours can be encoded in linear segments. The oscillations can be stored as segments with rate and depth parameters.

Data Chunk BODY

The BODY chunk contains the audio data samples.

```
#define ID_BODY MakeID('B', 'O', 'D', 'Y')

typedef character BYTE;     /* 8 bit signed number, -128 through 127. */
/* BODY chunk contains a BYTE[], array of audio data samples.         */
```

The BODY contains data samples grouped by octave. Within each octave are one-shot and repeat portions. Figure 2 depicts this arrangement of samples for an 8SVX where `oneShotHiSamples` = 24, `repeatHiSamples` = 16, `samplesPerHiCycle` = 8, and `ctOctave` = 3. The major divisions are octaves, the intermediate divisions separate the one-shot and repeat portions, and the minor divisions are cycles.

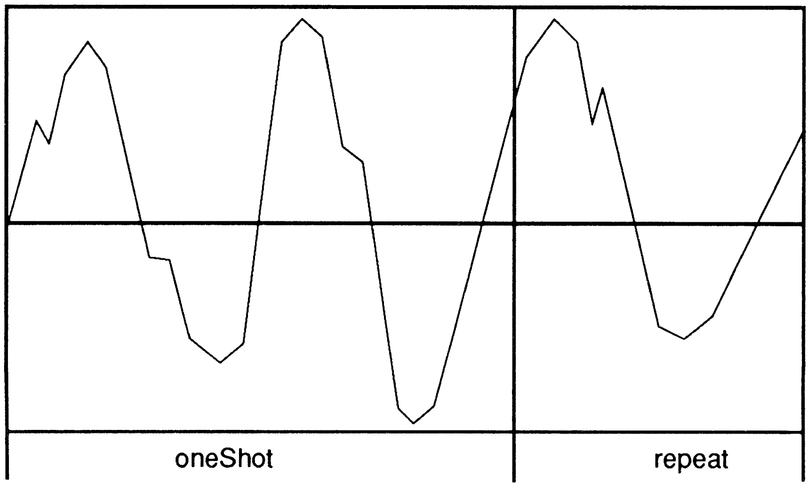


In general, the BODY has `ctOctave` octaves of data. The highest frequency octave comes first, comprising the fewest samples: `oneShotHiSamples + repeatHiSamples`. Each successive octave contains twice as many samples as the next higher octave but the same number of cycles. The lowest frequency octave comes last with the most samples: $2^{\text{ctOctave}-1}$ `* (oneShotHiSamples + repeatHiSamples)`.

The number of samples in the BODY chunk is

$$(2^0 + \ldots + 2^{\text{ctOctave}-1}) * (\texttt{oneShotHiSamples} + \texttt{repeatHiSamples})$$

Figure 3, below, looks closer at an example waveform within one octave of a different BODY chunk. In this example, `oneShotHiSamples / samplesPerHiCycle = 2` cycles and `repeatHiSamples / samplesPerHiCycle = 1` cycle.



To avoid playback "clicks" the one-shot part should begin with a small sample value, and flow smoothly into the repeat part. The end of the repeat part should flow smoothly into the beginning of the next repeat part.

If the VHDR field `sCompression` $\neq$ `sCmpNone`, the BODY chunk is just an array of data bytes to feed through the specified decompresser function. All this stuff about sample sizes, octaves, and repeat parts applies to the decompressed data.

Be sure to follow an odd-length BODY chunk with a 0 pad byte.

Other Chunks

Issue: In the future, we may define an optional chunk containing Fourier series coefficients for a repeating waveform. An editor for this kind of synthesized voice could modify the coefficients and regenerate the waveform.

See the IFF Registry and the Third-Party Specification section for details on additional 8SVX Chunks such as CHAN, PAN, SEQN and FADE.

Appendix A. Quick Reference

Type Definitions

```
#define ID_8SVX MakeID('8', 'S', 'V', 'X')
#define ID_VHDR MakeID('V', 'H', 'D', 'R')

typedef LONG Fixed;     /* A fixed-point value, 16 bits to the left of  *
                           the point and 16 to the right. A Fixed is a  *
                           number of 2^16ths, i.e., 65536ths.           */
#define Unity 0x10000L /* Unity = Fixed 1.0 = maximum volume           */


/* sCompression: Choice of compression algorithm.                      */
#define sCmpNone        0    /* not compressed                         */
#define sCmpFibDelta    1    /* Fibonacci-delta encoding (Appendix C) */
                             /* Can be more kinds in the future.       */

typedef struct {
    ULONG oneShotHiSamples, /* # samples in the high octave 1-shot part */
          repeatHiSamples,  /* # samples in the high octave repeat part */
          samplesPerHiCycle;/* # samples/cycle in high octave, else 0   */
    UWORD samplesPerSec;    /* data sampling rate                       */
    UBYTE ctOctave,         /* # octaves of waveforms                   */
          sCompression;     /* data compression technique used          */
    Fixed volume;           /* playback volume from 0 to Unity (full    *
                             * volume). Map this value into the output  *
                             * hardware's dynamic range.                */
    } Voice8Header;


#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the voice's name.                      */

#define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice.         */

#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the author's name.                     */

#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations.             */

#define ID_ATAK MakeID('A', 'T', 'A', 'K')
#define ID_RLSE MakeID('R', 'L', 'S', 'E')


typedef struct {
    UWORD duration; /* segment duration in milliseconds, > 0            */
    Fixed dest;     /* destination volume factor                        */
    } EGPoint;


/* ATAK and RLSE chunks contain an EGPoint[],piecewise-linear envelope. */
/* The envelope defines a function of time returning Fixed values. It's *
 * used to scale the nominal volume specified in the Voice8Header.      */

#define ID_BODY MakeID('B', 'O', 'D', 'Y')
typedef character BYTE;     /* 8 bit signed number, -128 through 127.   */
/* BODY chunk contains a BYTE[], array of audio data samples.           */
```

8SVX Regular Expression

Here's a regular expression summary of the FORM 8SVX syntax. This could be an IFF file or part of one.

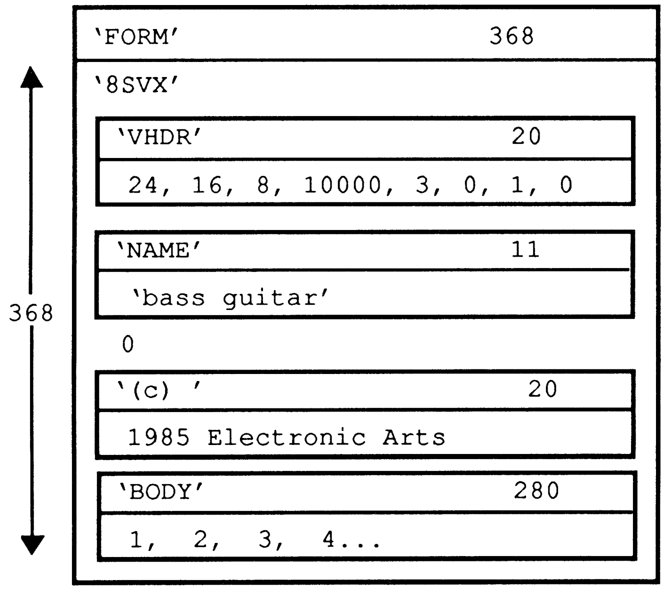
```
8SVX       ::= "FORM" #{  "8SVX" VHDR [NAME] [Copyright] [AUTH] ANNO*
                          [ATAK] [RLSE] BODY }

VHDR       ::= "VHDR" #{ Voice8Header     }
NAME       ::= "NAME" #{ CHAR*     } [0]
Copyright::= "(c) " #{ CHAR*     } [0]
AUTH       ::= "AUTH" #{ CHAR*     } [0]
ANNO       ::= "ANNO" #{ CHAR*     } [0]

ATAK       ::= "ATAK" #{ EGPoint* }
RLSE       ::= "RLSE" #{ EGPoint* }
BODY       ::= "FORM" #{ BYTE*     } [0]
```

The token "#" represents a `ckSize` LONG count of the following {braced} data bytes. E.g., a VHDR's "#" should equal `sizeof(Voice8Header)`. Literal items are shown in "quotes", [square bracket items] are optional, and "*" means 0 or more replications. A sometimes-needed pad byte is shown as "`[0]`".

Actually, the order of chunks in a FORM 8SVX is not as strict as this regular expression indicates. The property chunks VHDR, NAME, Copyright, and AUTH may actually appear in any order as long as they all precede the BODY chunk. The optional data chunks ANNO, ATAK, and RLSE don't have to precede the BODY chunk. And of course, new kinds of chunks may appear inside a FORM 8SVX in the future.

Appendix B. 8SVX Example

Here's a box diagram for a simple example containing the three octave BODY shown earlier in Figure 2.



The "0" after the NAME chunk is a pad byte.

Appendix C. Fibonacci Delta Compression

This is Steve Hayes' Fibonacci Delta sound compression technique. It's like the traditional delta encoding but encodes each delta in a mere 4 bits. The compressed data is half the size of the original data plus a 2-byte overhead for the initial value. This much compression introduces some distortion, so try it out and use it with discretion.

To achieve a reasonable slew rate, this algorithm looks up each stored 4-bit value in a table of Fibonacci numbers. So very small deltas are encoded precisely while larger deltas are approximated. When it has to make approximations, the compressor should adjust all the values (forwards and backwards in time) for minimum overall distortion.

Here is the decompressor written in the C programming language.

```
/* Fibonacci delta encoding for sound data. */
BYTE codeToDelta[16] = {-34,-21,-13,-8,-5,-3,-2,-1,0,1,2,3,5,8,13,21};

/* Unpack Fibonacci-delta encoded data from n byte source buffer into
 * 2*n byte dest buffer, given initial data value x.  It returns the
 * last data value x so you can call it several times to incrementally
 * decompress the data.                                              */
short D1Unpack(source, n, dest, x)
    BYTE source[], dest[];
    LONG n;
    BYTE x;
    {
    BYTE d;
    LONG i, lim;

    lim = n << 1;
    for (i = 0; i < lim; ++i)
            { /* Decode a data nibble; high nibble then low nibble. */
            d = source[i >> 1];     /* get a pair of nibbles        */
            if (i & 1)              /* select low or high nibble?   */
                    d &= 0xf;       /* mask to get the low nibble   */
            else
                    d >>= 4;        /* shift to get the high nibble */
            x += codeToDelta[d];    /* add in the decoded delta     */
            dest[i] = x;            /* store a 1-byte sample        */
            }
    return(x);
    }

/* Unpack Fibonacci-delta encoded data from n byte source buffer into
 * 2*(n-2) byte dest buffer. Source buffer has a pad byte, an 8-bit
 * initial value, followed by n-2 bytes comprising 2*(n-2) 4-bit
 * encoded samples.                                                  */

void DUnpack(source, n, dest)
    BYTE source[], dest[];
    LONG n;
    {
      D1Unpack(source + 2, n - 2, dest, source[1]);
    }
```

IFF FORM and Chunk Registry

This section contains the official list of registered FORM and Chunk names that are reserved and in use. This list is often referred to as the 3rd part registry since these are FORM and Chunk types created by application developers and not part of the original IFF specification created by Electronic Arts and Commodore.

For all FORM and Chunk types that are public, the official specifications from the third party company are listed (in alphabetical order). At the end of this section are additional documents describing how the ILBM FORM type works on the Amiga.

New chunks and FORMS should be registered with CATS US, IFF Registry, 1200 Wilson Drive, West Chester, PA. 19380. Please make all submissions on Amiga diskette and include your address, phone, and fax.

IFF Form and Chunk Registry Page 1

IFF FORM And Chunk Registry

The following is an alphabetical list of registered FORMs, generic chunks (shown as (any).chunkname), and registered new chunks for existing FORMs (shown as formname.chunkname).

The center column describes where additional information on the FORM or chunk may be found. Items marked "EA_IFF" are described in the main chapters of the EA IFF specs. Those marked "TP_SPECS" are described in the this section. Items marked "propos" are proposals which have been submitted to CATS, some of which are private. And items marked with "------" are private or as yet unreleased specifications.

New chunks and FORMS should be registered with CATS US, IFF Registry, 1200 Wilson Drive, West Chester, PA. 19380. Please make all submissions on Amiga diskette and include your address, phone, and fax.

(any).ANNO	EA_IFF	EA IFF 85 Generic Annotation chunk
(any).AUTH	EA_IFF	EA IFF 85 Generic Author chunk
(any).CHRS	EA_IFF	EA IFF 85 Generic character string chunk
(any).CSET.doc	IFF_TP	chunk for specifying character set
(any).FVER.doc	IFF_TP	chunk for 2.0 VERSION string of an IFF file
(any).HLID.doc	----	HotLink IDentification (Contact CATS for info)
(any).NAME	EA_IFF	EA IFF 85 Generic Name of art, music, etc. chunk
(any).TEXT	EA_IFF	EA IFF 85 Generic unformatted ASCII text chunk
(any).(c)	EA_IFF	EA IFF 85 Generic Copyright text chunk
8SVX	EA_IFF	EA IFF 85 8-bit sound sample form
8SVX.CHAN.PAN.doc	IFF_TP	Stereo chunks for 8SVX form
8SVX.SEQN.FADE.doc	IFF_TP	Looping chunks for 8SVX form
ACBM.doc	IFF_TP	Amiga Contiguous Bitmap form
AHAM	----	unregistered (???)
AIFF.doc	IFF_TP	Audio 1-32 bit samples (Mac,AppleII,Synthia Pro)
ANBM.doc	IFF_TP	Animated bitmap form (Framer, Deluxe Video)
ANIM.brush.doc	IFF_TP	ANIM brush format
ANIM.doc	IFF_TP	Cel animation form
ANIM.op6.doc	IFF_TP	Stereo (3D) Animations
ARC.proposal	propos	archive format proposal (old)
ARES	----	unregistered (???)
ATXT	----	temporariliy reserved
AVCF	----	AmigaVision flow (format not yet released)
AVCF.doc	IFF_TP	
AVCO	----	AmigaVision commands (format not yet released)
AVEV	----	AmigaVision events (format not yet released)
BANK	----	Soundquest Editor/Librarian MIDI Sysex dump
BBSD	----	BBS Database, F.Patnaude,Jr., Phalanx Software
C100	----	Cloanto Italia private format
CAT	EA_IFF	EA IFF 85 group identifier
CHBM	----	Chunky bitmap (name reserved by Eric Lavitsky)
CLIP	----	CAT CLIP to hold various formats in clipboard
CPFM	----	Cloanto Personal FontMaker (doc in their manual)
DCCL	----	DCCL - DCTV paint clip
DCPA	----	DCPA - DCTV paint palette
DCTV	----	DCTV - DCTV raw picture file
DECK	----	private format for Inovatronics CanDo
DR2D.doc	IFF_TP	2-D Object standard format
DRAW	----	reserved by Jim Bayless, 12/90
FANT.doc	IFF_TP	Fantavision movie format
FIGR	----	Deluxe Video - reserved
FNTR	EA_IFF	EA IFF 85 reserved for raster font
FNTV	EA_IFF	EA IFF 85 reserved for vector font
FORM	EA_IFF	EA IFF 85 group identifier

IFF Form and Chunk Registry Page 2

FTXT	EA_IFF	EA IFF 85 formatted text form
GRYP.proposal	propos	byteplane storage proposal (copyrighted)
GSCR	EA_IFF	EA IFF 85 reserved gen. music score
GUI.proposal	propos	user interface storage proposal (private)
HEAD.doc	IFF_TP	Flow - New Horizons Software
ILBM	EA_IFF	EA IFF 85 raster bitmap form
ILBM.3DCM	----	reserved by Haitex
ILBM.3DPA	----	reserved by Haitex
ILBM.ASDG	----	private ASDG application chunk
ILBM.BHBA	----	private Photon Paint chunk (brushes)
ILBM.BHCP	----	private Photon Paint chunk (screens)
ILBM.BHSM	----	private Photon Paint chunk
ILBM.CLUT.doc	IFF_TP	Color Lookup Table chunk
ILBM.CMYK.doc	IFF_TP	Cyan, Magenta, Yellow, & Black cmap (Contact CATS)
ILBM.CNAM.doc	----	Color naming chunk (Soft-Logik) (Contact CATS)
ILBM.CTBL.DYCP.doc	IFF_TP	Newtek Dynamic Ham color chunks
ILBM.DCTV	----	reserved
ILBM.DGVW	----	private Newtek DigiView chunk
ILBM.DPI.doc	IFF_TP	Dots per inch chunk
ILBM.DPPV.doc	IFF_TP	DPaint perspective chunk (EA)
ILBM.DRNG.doc	IFF_TP	DPaint IV enhanced color cycle chunk (EA)
ILBM.EPSF.doc	IFF_TP	Encapsulated Postscript chunk
ILBM.TMAP	----	Transparency map (temporarily reserved)
ILBM.VTAG.proposal	propos	Viewmode tags chunk suggestion
ILBM.XBMI.doc	IFF_TP	eXtended BitMap Information (Contact CATS)
IOBJ	----	reserved by Seven Seas Software
ITRF	----	reserved
LIST	EA_IFF	EA IFF 85 group identifier
MIDI	----	Circum Design
MOVI	----	LIST MOVI - private format
MSCX	----	private Music-X format
MSMP	----	temporarily reserved
MTRX.doc	IFF_TP	Numerical data storage (MathVision - Seven Seas)
NSEQ	----	Numerical sequence (Stockhausen GmbH)
OCMP	EA_IFF	EA IFF 85 reserved computer prop
OCPU	EA_IFF	EA IFF 85 reserved processor prop
OPGM	EA_IFF	EA IFF 85 reserved program prop
OSN	EA_IFF	EA IFF 85 reserved serial num prop
PGTB.doc	IFF_TP	Program traceback (SAS Institute)
PICS	EA_IFF	EA IFF 85 reserved Macintosh picture
PLBM	EA_IFF	EA IFF 85 reserved obsolete name
PROP	EA_IFF	EA IFF 85 group identifier
PRSP.doc	IFF_TP	DPaint IV perspective move form (EA)
PTCH	----	Patch file format (SAS Institute)
PTXT	----	temporarily reserved
README	----	
RGB4	----	4-bit RGB (format not available)
RGBN-RGB8.doc	IFF_TP	RGB image forms, Turbo Silver (Impulse)
RGBX	----	temporarily reserved
ROXN	----	private animation form
SAMP.doc	IFF_TP	Sampled sound format
SC3D	----	private scene format (Sculpt-3D)
SHAK	----	private Shakespeare format
SHO1	----	Reserved by Gary Bonham (private)
SHOW	----	Reserved by Gary Bonham (private)
SMUS	EA_IFF	EA IFF 85 simple music score form
SYTH	----	SoundQuest Master Librarian MIDI System driver
TCDE	----	reserved by Merging Technologies
TDDD.doc	IFF_TP	3-D rendering data, Turbo Silver (Impulse)
UNAM	EA_IFF	EA IFF 85 reserved user name prop
USCR	EA_IFF	EA IFF 85 reserved Uhuru score
UVOX	EA_IFF	EA IFF 85 reserved Uhuru Mac voice
VDEO	----	private Deluxe Video format
WORD.doc	IFF_TP	ProWrite document format (New Horizons)

0000.CSET.doc

```
chunk for specifying character set

Registered by Martin Taillefer.

A chunk for use in any FORM, to specify character set used for
        text in FORM.


struct CSet {
        LONG    CodeSet;        /* 0=ECMA Latin 1 (std Amiga charset) */
                                /* CBM will define additional values  */
        LONG    Reserved[7];
        }
```

0000.FVER.doc

```
chunk for 2.0 VERSION string of an IFF file

Registered by Martin Taillefer.

A chunk for use in any FORM, to contain standard 2.0 version string.

$VER: name ver.rev

where "name" is the name or identifier of the file
and ver.rev is a version/revision such as 37.1

Example:

$VER: workbench.catalog 37.42
```

```
Stereo chunks for 8SVX form

                    SMUS.CHAN and SMUS.PAN Chunks
             Stereo imaging in the "8SVX" IFF 8-bit Sample Voice
             --------------------------------------------------
                Registered by David Jones, Gold Disk Inc.
```

There are two ways to create stereo imaging when playing back a digitized sound. The first relies on the original sound being created with a stereo sampler: two different samples are digitized simultaneously, using right and left inputs. To play back this type of sample while maintaining the stereo imaging, both channels must be set to the same volume. The second type of stereo sound plays the identical information on two different channels at different volumes. This gives the sample an absolute position in the stereo field. Unfortunately, there are currently a number of methods for doing this currently implemented on the Amiga, none truly adhering to any type of standard. What I have tried to to is provide a way of doing this consistently, while retaining compatibility with existing (non-standard) systems. Introduced below are two optional data chunks, CHAN and PAN. CHAN deals with sounds sampled in stereo, and PAN with samples given stereo characteristics after the fact.

Optional Data Chunk CHAN

This chunk is already written by the software for a popular stereo sampler. To maintain the ability read these samples, its implementation here is therefore limited to maintain compatability.

The optional data chunk CHAN gives the information neccessary to play a sample on a specified channel, or combination of channels. This chunk would be useful for programs employing stereo recording or playback of sampled sounds.

```
        #define RIGHT           4L
        #define LEFT            2L
        #define STEREO          6L

        #define ID_CHAN MakeID('C','H','A','N')

        typedef sampletype LONG;
```

If "sampletype" is RIGHT, the program reading the sample knows that it was originally intended to play on a channel routed to the right speaker, (channels 1 and 2 on the Amiga). If "sampletype" is LEFT, the left speaker was intended (Amiga channels 0 and 3). It is left to the discretion of the programmer to decide whether or not to play a sample when a channel on the side designated by "sampletype" cannot be allocated.

If "sampletype" is STEREO, then the sample requires a pair of channels routed to both speakers (Amiga pairs [0,1] and [2,3]). The BODY chunk for stereo pairs contains both left and right information. To adhere to existing conventions, sampling software should write first the LEFT information, followed by the RIGHT. The LEFT and RIGHT information should be equal in length.

Again, it is left to the programmer to decide what to do if a channel for a stereo pair can't be allocated; wether to play the available channel only, or to allocate another channels routed to the wrong speaker.

Optional Data Chunk PAN

The optional data chunk PAN provides the neccessary information to create a stereo sound using a single array of data. It is neccessary to replay the sample simultaneously on two channels, at different volumes.

```
        #define ID_PAN MakeID('P','A','N',' ')

        typedef sposition Fixed; /* 0 <= sposition <= Unity */
                                                        /* Unity is elsewhere #def
ined as 10000L, and
                                                         * refers to the maximum p
ossible volume.
                                                         * /

        /* Please note that 'Fixed' (elsewhere #defined as LONG) is used to
         * allow for compatabilty between audio hardware of different resolutions.
         */
```

The 'sposition' variable describes a position in the stereo field. The numbers of discrete stereo positions available is equal to 1/2 the number of discrete volumes for a single channel.

The sample must be played on both the right and left channels. The overall volume of the sample is determined by the "volume" field in the Voice8Header structure in the VHDR chunk.

```
The left channel volume = overall volume / (Unity / sposition).
 "  right    "       "   = overall volume - left channel volume.

For example:
        If sposition = Unity, the sample is panned all the way to the left.
        If sposition = 0, the sample is panned all the way to the right.
        If sposition = Unity/2, the sample is centered in the stereo field.
```

Looping chunks for 8SVX form

SEQN and FADE Chunks

Multiple Loop Sequencing in the "8SVX" IFF 8-bit Sample Voice
----
Registered by Peter Norman, RamScan Software Pty Ltd.

Sound samples are notorious for demanding huge amounts of memory.

While earlier uses of digital sound on the Amiga were mainly in the form of short looping waveforms for use as musical instruments, many people today wish to record several seconds (even minutes) of sound. This of course eats memory.

Assuming that quite often the content of these recordings is music, and that quite often music contains several passages which repeat at given times, "verse1 .. chorus .. verse2 .. chorus .." etc, a useful extention has been added to the 8SVX list of optional data chunks. It's purpose is to conserve memory by having the computer repeat sections rather than having several instances of a similar sound or musical passage taking up valuable sample space.

The "SEQN" chunk has been created to define "Multiple" loops or sections within a single octave 8SVX MONO or STEREO waveform.

It is intended that a sampled sound player program which supports this chunk will play sections of the waveform sequentially in an order that the SEQN chunk specifies. This means for example, if an identical chorus repeats throughout a recording, rather than have this chorus stored several times along the waveform, it is only necessary to have one copy of the chorus stored in the waveform.

A "SEQeNce" of definitions can then be set up to have the computer loop back and repeat the chorus at the required time. The remaining choruses stored in the waveform will no longer be necessary and can be removed.

E.g., if we had a recording of the following example, we would find that there are several parts which simply repeat. Substantial savings can be made by having the computer repeat sections rather than have them stored in memory.

EXAMPLE

"Haaaallelujah....Haaaallelujah...Hallelujah..Hallelujah..Halleeeelujaaaah."

Applying a sequence to the above recording would look as follows.

```
Haaaallelujah....Haaaallelujah...Hallelujah..Hallelujah..Halleeeelujaaaah.
[     Loop1     ]
[     Loop2     ]
                                  [  Loop3   ]
                                  [  Loop4   ]
                                                      [      Loop5      ]
               [   Dead Space   ]          [ Dead Space ]
```

The DEAD SPACE can be removed. With careful editing of the multiple loop positions, the passage can be made to sound exactly the same as the original with far less memory required.

Chunk Definitions...

Optional Data Chunk SEQN
____

The optional data chunk SEQN gives the information necessary to play a sample in a sequence of defined blocks. To have a segment repeat twice, the definition occurs twice in the list.

This list consists of pairs of ULONG "loop start" and "end" definitions which are offsets from the start of the waveform. The locations or values must be LONGWORD aligned (divisable by 4).

To determine how many loop definitions in a given file, simply divide the SEQN chunk size by 8.

E.g., if chunk size == 40 ... number of loops = (40 / 8) .. equals 5 loops.

The raw data in a file might look like this...

```
'S-E-Q-N' [ size ] [     Loop 1     ] [     Loop 2     ] [     Loop 3     ]

 5345514E 00000028 00000000 00000C00 00000000 00000C00 00000C08 00002000
             ^
             ^     'Haaaallelujah..' 'Haaaallelujah..'   'Hallelujah..'
             ^
             ^
             40 bytes decimal / 8 = 5 loop or segments


      [     Loop 4     ] [     Loop 5     ]'B-O-D-Y'   Size     Data

      00000C08 00002000 00002008 00003000 424F4459 000BE974 010101010101010

       'Hallelujah..'   'Halleeeelujah..'
```

In a waveform containing SEQN chunks, the oneShotHiSamples should be set to 0 and the repeatHiSamples should equal the BODY length (divided by 2 if STEREO).

Remember the locations of the start and end of each segment or loop should be LONGWORD aligned.

If the waveform is Stereo, treat the values and locations in exactly the same way. In other words, if a loop starts at location 400 within a Stereo waveform, you start the sound at the 400th byte position in the left data and the 400th byte position in the right data simultaneously.

```
#define ID_SEQN MakeID('S','E','Q','N')
```

Optional Data Chunk FADE

The FADE chunk defines at what loop number the sound should begin to fade away to silence. It is possible to finish a sample of music in much the same way as commercial music does today. A FADE chunk consists of one ULONG value which has a number in it. This number corresponds to the loop number at which the fade should begin.

eg. You may have a waveform containing 50 loops. A FADE definition of 45 will specify that once loop 45 is reached, fading to zero volume should begin. The rate at which this fade takes place is determined by the length of time left to play. The playing software should do a calculation based on the following...

Length of all remaining sequences including current sequence (in bytes)

divided by

the current playback rate in samples per second

= time remaining.

Begin stepping the volume down at a rate which will hit zero volume just as the waveform finishes.

The raw data in a file may look like this.

```
 'F-A-D-E'  [ Size ]   Loop No.  'B-O-D-Y'   Size   Data..

  46414445  00000004   0000002D   424F4459 000BE974 01010101 01010101 etc etc
                          ^
                       Start fading when loop number 45 is reached.


        #define ID_FADE MakeID('F','A','D','E')
```

Although order shouldn't make much difference, it is a general rule of thumb that SEQN should come before FADE and FADE should be last before the BODY.

Stereo waveforms would have CHAN,SEQN,FADE,BODY in that order.

```
Amiga Contiguous Bitmap form

IFF FORM / CHUNK DESCRIPTION
============================

Form/Chunk ID:    FORM  ACBM  (Amiga Contiguous BitMap)
                  Chunk ABIT  (Amiga BITplanes)

Date Submitted:   05/29/86
Submitted by:     Carolyn Scheppner   CBM


FORM
====

FORM ID:  ACBM  (Amiga Contiguous BitMap)

FORM Description:

   FORM ACBM has the same format as FORM ILBM except the normal BODY
chunk (InterLeaved BitMap) is replaced by an ABIT chunk (Amiga BITplanes).

FORM Purpose:

   To enable faster loading/saving of screens, especially from Basic,
while retaining the flexibility and portability of IFF format files.


CHUNKS
======

Chunk ID:   ABIT  (Amiga BITplanes)

Chunk Description:

   The ABIT chunk contains contiguous bitplane data.  The chunk contains
sequential data for bitplane 0 through bitplane n.

Chunk Purpose:

   To enable loading/storing of bitmaps with one DOS Read/Write per
bitplane.  Significant speed increases are realized when loading/saving
screens from Basic.


SUPPORTING SOFTWARE
===================

(Public Domain, available soon via Fish PD disk, various networks)

LoadILBM-SaveACBM (AmigaBasic)
   Loads and displays an IFF ILBM pic file (Graphicraft, DPaint, Images).
   Optionally saves the screen in ACBM format.

LoadACBM (AmigaBasic)
   Loads and display an ACBM format pic file.

SaveILBM (AmigaBasic)
   Saves a demo screen as an ILBM pic file which can be loaded into
   Graphicraft, DPaint, Images.
```

Audio 1-32 bit samples (Mac,AppleII,Synthia Pro)

provided by Steve Milne and Matt Deatherage, Apple Computer, Inc.

AIFF: Audio Interchange File Format File

The Audio Interchange File Format (Audio IFF) provides a standard for storing sampled sounds. The format is quite flexible, allowing the storage of monaural or multichannel sampled sounds at a variety of sample rates and sample widths.

Audio IFF conforms to the "'EA IFF 85' Standard for Interchange Format Files" developed by Electronic Arts.

Audio IFF is primarily an interchange format, although application designers should find it flexible enough to use as a data storage format as well. If an application does choose to use a different storage format, it should be able to convert to and from the format defined in this document. This ability to convert will facilitate the sharing of sound data between applications.

Audio IFF is the result of several meetings held with music developers over a period of ten months during 1987 and 1988. Apple Computer greatly appreciates the comments and cooperation provided by all developers who helped define this standard.

Another "EA IFF 85" sound storage format is "'8SVX' IFF 8-bit Sampled Voice", by Electronic Arts. "8SVX," which handles eight-bit monaural samples, is intended mainly for storing sound for playback on personal computers. Audio IFF is intended for use with a larger variety of computers, sampled sound instruments, sound software applications, and high fidelity recording devices.

Data Types

A C-like language will be used to describe the data structures in this document The data types used are listed below.

char:	8 bits signed. A char can contain more than just ASCII characters. It can contain any number from -128 to 127 (inclusive).
unsigned char:	8 bits signed. Contains any number from 0 to 255 (inclusive).
short:	16 bits signed. Contains any number from -32,768 to 32,767 (inclusive).
unsigned short:	16 bits unsigned. Contains any number from 0 to 65,535 (inclusive).
long:	32 bits signed. Contains any number from -2,147,483,648 to 2,147,483,647 (inclusive).
unsigned long:	32 bits unsigned. Contains any number from 0 to 4,294,967,295 (inclusive).
extended:	80 bit IEEE Standard 754 floating point number (Standard Apple Numeric Environment [SANE] data type Extended)
pstring:	Pascal-style string, a one-byte count followed by text bytes. The total number of bytes in this data type should be even. A pad byte can be added to the end of the text to accomplish this. This pad byte is not reflected in the count.
ID:	32 bits, the concatenation of four printable ASCII characters in the range " " (space, 0x20) through "~" (tilde, 0x7E). Leading spaces are not allowed in the ID but trailing spaces are OK. Control characters are forbidden.

Constants

Decimal values are referred to as a string of digits, for example 123, 0, 100 are all decimal numbers. Hexadecimal values are preceded by a 0x - e.g., 0x0A, 0x1, 0x64.

Data Organization

All data is stored in Motorola 68000 format. The bytes of multiple-byte values are stored with the high-order bytes first. Data is organized as follows:

```
               7  6  5  4  3  2  1  0
             +-----------------------+
      char:  | msb               lsb |
             +-----------------------+

              15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
             +-----------------------+-----------------------+
      char:  | msb    byte 0         |          byte 1   lsb |
             +-----------------------+-----------------------+

              15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
             +-----------------------+-----------------------+
      char:  | msb    byte 0         |          byte 1       |
             +-----------------------+-----------------------+
      char:  |        byte 2         |          byte 3   lsb |
             +-----------------------+-----------------------+
```

Figure 1: IFF data storage formats

Referring to Audio IFF

The official name for this standard is Audio Interchange File Format. If an application program needs to present the name of this format to a user, such as in a "Save As..." dialog box, the name can be abbreviated to Audio IFF. Referring to Audio IFF files by a four-letter abbreviation (i.e., "AIFF") in user-level documentation or program-generated messages should be avoided.

File Structure

The "'EA IFF 85' Standard for Interchange Format Files" defines an overall structure for storing data in files. Audio IFF conforms to those portions of "EA IFF 85" that are germane to Audio IFF. For a more complete discussion of "EA IFF 85", please refer to the document "'EAIFF 85', Standard for Interchange Format Files."

An "EA IFF 85" file is made up of a number of chunks of data. Chunks are the building blocks of "EA IFF 85" files. A chunk consists of some header information followed by data:

```
      +--------------------+
      |       ckID         |\
      +--------------------+ } header info
      |      ckSize        |/
      +--------------------+
      |                    |
      |                    |
      |       data         |
      |                    |
      |                    |
      +--------------------+
```

Figure 2: IFF Chunk structure

A chunk can be represented using our C-like language in the following manner:

```
    typedef struct {
        ID              ckID;           /* chunk ID             */
        long            ckSize;         /* chunk Size           */

        char            ckData[];       /* data                 */
        } Chunk;
```

The ckID describes the format of the data portion of a chunk. A program can determine how to interpret the chunk data by examining ckID.

The ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize.

The ckData contains the data stored in the chunk. The format of this data is determined by ckID. If the data is an odd number of bytes in length, a zero pad byte must be added at the end. The pad byte is not included in ckSize.

Note that an array with no size specification (e.g., char ckData[];) indicates a variable-sized array in our C-like language. This differs from standard C.

An Audio IFF file is a collection of a number of different types of chunks. There is a Common Chunk which contains important parameters describing the sampled sound, such as its length and sample rate. There is a Sound Data Chunk which contains the actual audio samples. There are several other optional chunks which define markers, list instrument parameters, store application-specific information, etc. All of these chunks are described in detail in later sections of this document.

The chunks in an Audio IFF file are grouped together in a container chunk. "EA IFF 85" Standard for Interchange Format Files defines a number of container chunks, but the one used by Audio IFF is called a FORM. A FORM has the following format:

```
    typedef struct {
        ID      ckID;
        long    ckSize;

        ID      formType;
        char    chunks[];
        }
```

The ckID is always 'FORM'. This indicates that this is a FORM chunk.

The ckSize contains the size of data portion of the 'FORM' chunk. Note that the data portion has been broken into two parts, formType and chunks[].

The formType field describes what's in the 'FORM' chunk. For Audio IFF files, formType is always 'AIFF'. This indicates that the chunks within the FORM pertain to sampled sound. A FORM chunk of formType 'AIFF' is called a FORM AIFF.

The chunks field are the chunks contained within the FORM. These chunks are called local chunks. A FORM AIFF along with its local chunks make up an Audio IFF file.

Here is an example of a simple Audio IFF file. It consists of a file containing single FORM AIFF which contains two local chunks, a Common Chunk and a Sound Data Chunk.

```
 ____________________________
| FORM AIFF Chunk            |
|   ckID  = 'FORM'           |
|   formType = 'AIFF'        |
|    __________________      |
|   | Common Chunk     |     |
|   |   ckID = 'COMM'  |     |
|   |__________________|     |
|    __________________      |
|   | Sound Data Chunk |     |
|   |   ckID = 'SSND'  |     |
|   |__________________|     |
|____________________________|
```

Figure 3: Simple Audio IFF File

There are no restrictions on the ordering of local chunks within a FORM AIFF.

A more detailed example of an Audio IFF file can be found in Appendix A. Please refer to this example as often as necessary while reading the remainder of this document.

Storage of AIFF on Apple and Other Platforms

On a Macintosh, the FORM AIFF, is stored in the data fork of an Audio IFF file. The Macintosh file type of an Audio IFF file is 'AIFF'. This is the same as the formType of the FORM AIFF. Macintosh applications should not store any information in Audio IFF file's resource fork, as this information may not be preserved by all applications. Applications can use the Application Specific Chunk, defined later in this document, to store extra information specific to their application.

Audio IFF files may be identified in other Apple file systems as well. On a Macintosh under MFS or HFS, the FORM AIFF is stored in the data fork of a file with file type "AIFF." This is the same as the formType of the FORM AIFF.

On an operating system such as MS-DOS or UNIX, where it is customary to use a file name extension, it is recommended that Audio IFF file names use ".AIF" for the extension.

On an Apple II, FORM AIFF is stored in a file with file type $D8 and auxiliary type $0000. Versions 1.2 and earlier of the Audio IFF standard used file type $CB and auxiliary type $0000. This is incorrect; the assignment listed in this document is the correct assignment.

On the Apple IIGS stereo data is stored with right data on even channels and left data on odd channels. Some portions of AIFF do not follow this convention. Even where it does follow the convention, AIFF usually uses channel two for right data instead of channel zero as most Apple IIGS standards do. Be prepared to interpret data accordingly.

Local Chunk Types

The formats of the different local chunk types found within a FORM AIFF are described in the following sections, as are their ckIDs.

There are two types of chunks: required and optional. The Common Chunk is required. The Sound Data chunk is required if the sampled sound has a length greater than zero. All other chunks are optional. All applications that use FORM AIFF must be able to read the required chunks and can choose to selectively ignore the optional chunks. A program that copies a FORM AIFF should copy all the chunks in the FORM AIFF, even those it chooses not to interpret.

The Common Chunk

The Common Chunk describes fundamental parameters of the sampled sound.

```
    #define     CommonID        'COMM'  /* ckID for Common Chunk */

    typedef struct {
        ID              ckID;
        long            ckSize;

        short           numChannels;
        unsigned long   numSampleFrames;
        short           sampleSize;
        extended        sampleRate;
    } CommonChunk;
```

The ckID is always 'COMM'. The ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize. For the Common Chunk, ckSize is always 18.

The numChannels field contains the number of audio channels for the sound. A value of 1 means monophonic sound, 2 means stereo, and 4 means four channel sound, etc. Any number of audio channels may be represented. For multichannel sounds, single sample points from each channel are interleaved. A set of interleaved sample points is called a sample frame.

The actual sound samples are stored in another chunk, the Sound Data Chunk, which will be described shortly.

Single sample points from each channel are interleaved such that each sample frame is a sample point from the same moment in time for each channel available.

The numSampleFrames field contains the number of sample frames. This is not necessarily the same as the number of bytes nor the number of samplepoints in the Sound Data Chunk. The total number of sample points in the file is numSampleFrames times numChannels.

The sampleSize is the number of bits in each sample point. It can be any number from 1 to 32. The format of a sample point will be described in the next section.

The sampleRate field is the sample rate at which the sound is to be played back in sample frames per second.

One, and only one, Common Chunk is required in every FORM AIFF.

Sound Data Chunk

The Sound Data Chunk contains the actual sample frames.

```
    #define     SoundDataID     'SSND'  /* ckID for Sound Data Chunk    */

    typedef struct {
        ID              ckID;
        long            ckSize;

        unsigned long   offset;
        unsigned long   blockSize;
        unsigned char   SoundData [];
    }  SoundDataChunk;
```

The ckID is always 'SSND'. The ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize.

The offset field determines where the first sample frame in the soundData starts. The offset is in bytes. Most applications won't use offset and should set it to zero. Use for a non-zero offset is explained in the Block-Aligning Sound Data section below.

The blockSize is used in conjunction with offset for block-aligning sound data. It contains the size in bytes of the blocks that sound data is aligned to. As with offset, most applications won't use blockSize and should set it to zero. More information on blockSize is in the Block-Aligning Sound Data section below.

The soundData field contains the sample frames that make up the sound. The number of sample frames in the soundData is determined by the numSampleFrames field in the Common Chunk. Sample points and sample frames are explained in detail in the next section.

The Sound Data Chunk is required unless the numSampleFrames field in the Common Chunk is zero. A maximum of one Sound Data Chunk may appear in a FORM AIFF.

Sample Points and Sample Frames

A large part of interpreting Audio IFF files revolves around the two concepts of sample points and sample frames.

A sample point is a value representing a sample of a sound at a given point in time. Each sample point is stored as a linear, 2's-complement value which may be from 1 to 32 bits wide, as determined by sampleSize in the Common Chunk.

Sample points are stored in an integral number of contiguous bytes. One- to eight-bit wide sample points are stored in one byte, 9- to 16-bit wide sample points are stored in two bytes, 17- to 24-bit wide sample points are stored in three bytes, and 25- to 32-bit wide sample points are stored in four bytes (most significant byte first). When the width of a sample point is not a multiple of eight bits, the sample point data is left justified, with the remaining bits zeroed. An example case is illustrated in Figure 4. A 12-bit sample point, binary 101000010111, is stored left justified in two bytes. The remaining bits are set to zero.

```
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1   0   1   0   0   0   0   1 | 0   1   1   1   0   0   0   0 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
 <---------------------------------------------> <------------->
       12 bit sample point is left justified          rightmost
                                                      4 bits are
                                                      zero padded
```

Figure 4: 12-Bit Sample Point

For multichannel sounds, single sample points from each channel are interleaved. A set of interleaved sample points is called a sample frame. Single sample points from each channel are interleaved such that each sample frame is a sample point from the same moment in time for each channel available. This is illustrated in Figure 5 for the stereo (two channel) case.

```
         sample       sample               sample
         frame 0      frame 1              frame N
     _____ _____ _____ _____         _____ _____
    | ch1 | ch2 | ch1 | ch2 | . . . | ch1 | ch2 |
    |_____|_____|_____|_____|       |_____|_____|

                 _____
                |     | = one sample point
                |_____|
```

Figure 5: Sample Frames for Multichannel Sound

For monophonic sound, a sample frame is a single sample point. For multichannel sounds, you should follow the conventions in Figure 6.

	channel 1	2	3	4	5	6
stereo	left	right				
3 channel	left	right	center			
quad	front left	front right	rear left	rear right		
4 channel	left	center	right	surround		
6 channel	left	left center	center	right	right center	surround

Figure 6: Sample Frame Conventions for Multichannel Sound

Sample frames are stored contiguously in order of increasing time. The sample points within a sample frame are packed together; there are no unused bytes between them. Likewise, the sample frames are packed together with no pad bytes.

Block-Aligning Sound Data

There may be some applications that, to ensure real time recording and playback of audio, wish to align sampled sound data with fixed-size blocks. This alignment can be accomplished with the offset and blockSize parameters of the Sound Data Chunk, as shown in Figure 7.

```
     |\\ unused \\|          sample frames           |\\ unused \\|
     |____________|__________________________________|____________|
    <-- offset --><- numSampleFrames sample frames ->

 |  blockSize    |               |               |               |
 |<- bytes     ->|               |               |               |
 |_______________|_______________|_______________|_______________|
    block N-1        block N        block N+1       block N+2
```

Figure 7: Block-Aligned Sound Data

In Figure 7, the first sample frame starts at the beginning of block N. This is accomplished by skipping the first offset bytes of the soundData. Note too, that the soundData bytes can extend beyond valid sample frames, allowing the soundData bytes to end on a block boundary as well.

The blockSize specifies the size in bytes of the block to which you would align the sound data. A blockSize of zero indicates that the sound data does not need to be block-aligned. Applications that don't care about block alignment should set the blockSize and offset to zero when creating Audio IFF files. Applications that write block-aligned sound data should set blockSize to the appropriate block size. Applications that modify an existing Audio IFF file should try to preserve alignment of the sound data, although this is not required. If an application does not preserve alignment, it should set the blockSize and offset to zero. If an application needs to realign sound data to a different sized block, it should update blockSize and offset accordingly.

The Marker Chunk

The Marker Chunk contains markers that point to positions in the sound data. Markers can be used for whatever purposes an application desires. The Instrument Chunk, defined later in this Note, uses markers to mark loop beginning and end points.

Markers

A marker has the following format.

```
    typedef     short           MarkerId;

    typedef     struct  {
                MarkerID        id;
                unsigned long   position;
                pstring         markerName;
    } Marker;
```

The id is a number that uniquely identifies that marker within a FORM AIFF. The id can be any positive non-zero integer, as long as no other marker within the same FORM AIFF has the same id.

The marker's position in the sound data is determined by the position field. Markers conceptually fall between two sample frames. A marker that falls before the first sample frame in the sound data is at position zero, while a marker that falls between the first and second sample frame in the sound data is at position 1. Note that the units for position are sample frames, not bytes nor sample points.

```
                    Sample Frames
         ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
        |   |   |   |   |   |   |   |   |   |   |   |   |
        |___|___|___|___|___|___|___|___|___|___|___|___|
        ^                   ^                           ^
   position 0          position 5                 position 12
```

Figure 8: Sample Frame Marker Positions

The markerName field is a Pascal-style text string containing the name of the mark.

Note: Some "EA IFF 85" files store strings a C-strings (text bytes followed by a null terminating character) instead of Pascal-style strings. Audio IFF uses pstrings because they are more efficiently skipped over when scanning through chunks. Using pstrings, a program can skip over a string by adding the string count to the address of the first character. C strings require that each character in the string be examined for the null terminator.

Marker Chunk Format

The format for the data within a Marker Chunk is shown below.

```
    #define     MarkerID        'MARK'  /* ckID for Marker Chunk */

    typedef  struct {
        ID                          ckID;
        long                        ckSize;

        unsigned short          numMarkers;
        Marker                  Markers [];
    } MarkerChunk;
```

The ckID is always 'MARK'. The ckSize is the size of the data portion of the chunk in bytes. It does not include the 8 bytes used by ckID and ckSize.

The numMarkers field is the number of markers in the Marker Chunk. If numMarkers is non-zero, it is followed by the markers themselves. Because all fields in a marker are an even number of bytes, the length of any marker will always be even. Thus, markers are packed together with no unused bytes between them. The markers need not be ordered in any particular manner.

The Marker Chunk is optional. No more than one Marker Chunk can appear in a FORM AIFF.

The Instrument Chunk

The Instrument Chunk defines basic parameters that an instrument, such as a sample, could use to play the sound data.

Looping

Sound data can be looped, allowing a portion of the sound to be repeated in order to lengthen the sound. The structure below describes a loop.

```
    typedef struct {
        short   PlayMode;
        MarkerId beginLoop;
        MarkerId endLoop;
    } Loop;
```

A loop is marked with two points, a begin position and an end position. There are two ways to play a loop, forward looping and forward/backward looping. In the case of forward looping, playback begins at the beginning of the sound, continues past the begin position and continues to the end position, at which point playback starts again at the begin position. The segment between the begin and end positions, called the loop segment, is played repeatedly until interrupted by a user action, such as the release of a key on a sampling instrument.

```
    sample frames |   |   |   |<--- loop segment ---->|   |   |   |
                  |___|___|___|___|___|___|___|___|___|___|___|___|
                              ^                       ^
                       begin position            end position
```

Figure 9: Sample Frame Looping

With forward/backward looping, the loop segment is first played from the begin position to the end position, and then played backwards from the end position to the begin position. This flip-flop pattern is repeated over and over again until interrupted.

The playMode specifies which type of looping is to be performed:

```
        #define NoLooping               0
        #define ForwardLooping          1
        #define ForwardBackwardLooping  2
```

If NoLooping is specified, then the loop points are ignored during playback.

The beginLoop is a marker id that marks the begin position of the loop segment.

The endLoop marks the end position of a loop. The begin position must be less than the end position. If this is not the case, then the loop segment has zero or negative length and no looping takes place.

The Instrument Chunk Format

The format of the data within an Instrument Chunk is described below.

```
    #define             InstrumentID    'INST'  /*ckID for Instruments Chunk */

    typedef struct {
        ID              ckID;
        long            ckSize;

        char            baseNote;
        char            detune;
        char            lowNote;
        char            highNote;
        char            lowvelocity;
        char            highvelocity;
        short           gain;
        Loop            sustainLoop;
        Loop            releaseLoop;
    } InstrumentChunk;
```

The ckID is always 'INST'. ckSize is the size of the data portion of the chunk, in bytes. For the Instrument Chunk, ckSize is always 20.

The baseNote is the note at which the instrument plays back the sound data without pitch modification. Units are MIDI (MIDI is an acronym for Musical Instrument Digital Interface) note numbers, and are in the range 0 through 127. Middle C is 60.

The detune field determines how much the instrument should alter the pitch of the sound when it is played back. Units are in cents (1/100 of a semitone) and range from -50 to +50. Negative numbers mean that the pitch of the sound should be lowered, while positive numbers mean that it should be raised.

The lowNote and highNote fields specify the suggested range on a keyboard for playback of the sound data. The sound data should be played if the instrument is requested to play a note between the low and high, inclusive. The base note does not have to be within this range. Units for lowNote and highNote are MIDI note values.

The lowVelocity and highVelocity fields specify the suggested range of velocities for playback of the sound data. The sound data should be played if the note-on velocity is between low and high velocity, inclusive. Units are MIDI velocity values, 1 (lowest velocity) through 127 (highest velocity).

The gain is the amount by which to change the gain of the sound when it is played. Units are decibels. For example, 0db means no change, 6db means double the value of each sample point, while -6db means halve the value of each sample point.

The sustainLoop field specifies a loop that is to be played when an instrument is sustaining a sound.

The releaseLoop field specifies a loop that is to be played when an instrument is in the release phase of playing back a sound. The release phase usually occurs after a key on an instrument is released.

The Instrument Chunk is optional. No more than one Instrument Chunk can appear in a FORM AIFF.

ASIF Note: The Apple IIGS Sampled Instrument Format also defines a chunk with ID of "INST," which is not the same as the Audio IFF Instrument Chunk. A good way to tell the two chunks apart in generic IFF-style readers is by the ckSize fields.

The Audio IFF Instrument Chunk's ckSize field is always 20, whereas the Apple IIGS Sampled Instrument Format Instrument Chunk's ckSize field, for structural reasons, can never be 20.

The MIDI Data Chunk

The MIDI Data Chunk can be used to store MIDI data. Please refer to Musical Instrument Digital Interface Specification 1.0, available from the International MIDI Association, for more details on MIDI.

The primary purpose of this chunk is to store MIDI System Exclusive messages, although other types of MIDI data can be stored in the block as well. As more instruments come to market, they will likely have parameters that have not been included in the Audio IFF specification. The MIDI System Exclusive messages for these instruments may contain many parameters that are not included in the Instrument Chunk. For example, a new sampling instrument may have more than the two loops defined in the Instrument Chunk. These loops will likely be represented in the MIDI System Exclusive message for the new machine. This MIDI System Exclusive message can be stored in the MIDI Data Chunk.

```
    #define            MIDIDataID      'MIDI' /* ckID for MIDI Data Chunk */

    typedef struct {
        ID             ckID;
        long           ckSize;

        unsigned char  MIDIdata[];
    } MIDIDataChunk;
```

The ckID is always 'MIDI'. ckSize of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize.

The MIDIData field contains a stream of MIDI data.

The MIDI Data Chunk is optional. Any number of MIDI Data Chunks may exist in a FORM AIFF. If MIDI System Exclusive messages for several instruments are to be stored in a FORM AIFF, it is better to use one MIDI Data Chunk per instrument than one big MIDI Data Chunk for all of the instruments.

The Audio Recording Chunk

The Audio Recording Chunk contains information pertinent to audio recording devices.

```
    #define     AudioRecordingID 'AESD'         /* ckID for Audio Recording */
                                                /*  Chunk.                  */
    typedef struct {
        ID                      ckID
        long                    ckSize;

        unsigned char   AESChannelStatusData[24];
    } AudioRecordingChunk;
```

The ckID is always 'AESD'. The ckSize is the size of the data portion of the chunk, in bytes For the Audio Recording Chunk, ckSize is always 24.

The 24 bytes of AESCChannelStatusData are specified in the "AES Recommended Practice for Digital Audio Engineering - Serial Transmission Format for Linearly Represented Digital Audio Data", transmission of digital audio between audio devices. This information is duplicated in the Audio Recording Chunk for convenience. Of general interest would be bits 2, 3, and 4 of byte 0, which describe recording emphasis.

The Audio Recording Chunk is optional. No more than one Audio Recording Chunk may appear in a FORM AIFF.

The Application Specific Chunk

The Application Specific Chunk can be used for any purposes whatsoever by developers and application authors. For example, an application that edits sounds might want to use this chunk to store editor state parameters such as magnification levels, last cursor position, etc.

```
    #define     ApplicationSpecificID 'APPL' /* ckID for Application */
                                             /*  Specific Chunk.     */
    typedef struct {
        ID              ckID;
        long            ckSize;

        OSType          applicationSignature;
        char            data[];
    } ApplicationSpecificChunk;
```

The ckID is always 'APPL'. The ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize.

The applicationSignature identifies a particular application. For Macintosh applications, this will be the application's four character signature.

The OSType field is used by applications which run on platforms from Apple Computer, Inc. For the Apple II, the OStype field should be set to 'pdos'. For the Macintosh, this field should be set to the four character signature as registered with Apple Technical Support.

The data field is the data specific to the application.

The Application Specific Chunk is optional. Any number of Application Specific Chunks may exist in a single FORM AIFF.

The Comments Chunk

The Comments Chunk is used to store comments in the FORM AIFF. "EA IFF 85" has an Annotation Chunk (used in ASIF) that can be used for comments, but the Comments Chunk has two features not found in the "EA IFF 85" chunk. They are a time-stamp for the comment and a link to a marker.

Comment

A comment consists of a time stamp, marker id, and a text count followed by text.

```
    typedef struct {
        unsigned long   timeStamp;
        MarkerID        marker;
        unsigned short  count;
        char            text;
    } Comment;
```

The timeStamp indicates when the comment was created. On the Amiga, units are the number of seconds since January 1, 1978. On the Macintosh, units are the number of seconds since January 1, 1904.

A comment can be linked to a marker. This allows applications to store long descriptions of markers as a comment. If the comment is referring to a marker, then the marker field is the ID of that marker. Otherwise, marker is zero, indicating that this comment is not linked to a marker.

The count is the length of the text that makes up the comment. This is a 16-bit quantity, allowing much longer comments than would be available with a pstring.

The text field contains the comment itself.

The Comments Chunk is optional. No more than one Comments Chunk may appear in a single FORM AIFF.

Comments Chunk Format

```
    #define     CommentID       'COMT'  /* ckID for Comments Chunk  */

    typedef struct {
        ID              ckID;
        long            ckSize;

        unsigned short  numComments;
        Comment         comments[];
    }CommentsChunk;
```

The ckID is always 'COMT'. The ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize.

The numComments field contains the number of comments in the Comments Chunk. This is followed by the comments themselves. Comments are always even numbers of bytes in length, so there is no padding between comments in the Comments Chunk.

The Comments Chunk is optional. No more than one Comments Chunk may appear in a single FORM AIFF.

The Text Chunks, Name, Author, Copyright, Annotation

These four chunks are included in the definition of every "EA IFF 85" file. All are text chunks; their data portion consists solely of text. Each of these chunks is optional.

```
    #define     NameID 'NAME'   /* ckID for Name Chunk */
    #define     NameID 'AUTH'   /* ckID for Author Chunk */
    #define     NameID '(c) '   /* ckID for Copyright Chunk */
    #define     NameID 'ANNO'   /* ckID for Annotation Chunk */
```

```
    typedef struct {
        ID      ckID;
        long    ckSize;
        char    text[];
    }TextChunk;
```

The ckID is either 'NAME', 'AUTH', '(c) ', or 'ANNO' depending on whether the chunk is a Name Chunk, Author Chunk, Copyright Chunk, or Annotation Chunk, respectively. For the Copyright Chunk, the 'c' is lowercase and there is a space (0x20) after the close parenthesis.

The ckSize is the size of the data portion of the chunk, in this case the text.

The text field contains pure ASCII characters. it is not a pstring or a C string. The number of characters in text is determined by ckSize. The contents of text depend on the chunk, as described below:

Name Chunk. The text contains the name of the sampled sound. The Name Chunk is optional. No more than one Name Chunk may exist within a FORM AIFF.

Author Chunk. The text contains one or more author names. An author in this case is the creator of a sampled sound. The Author Chunk is optional. No more than one Author Chunk may exist within a FORM AIFF.

Copyright Chunk. The Copyright Chunk contains a copyright notice for the sound. The text field contains a date followed by the name of the copyright owner. The chunk ID '(c) ' serves as the copyright character. For example, a Copyright Chunk containing the text "1991 Commodore-Amiga, Inc." means "(c) 1991 Commodore-Amiga, Inc." The Copyright Chunk is optional. No more than one Copyright Chunk may exist within a FORM AIFF.

Annotation Chunk. The text contains a comment. Use of this chunk is discouraged within a FORM AIFF. The more powerful Comments Chunk should be used instead. The Annotation Chunk is optional. Many Annotation Chunks may exist within a FORM AIFF.

Chunk Precedence

Several of the local chunks for FORM AIFF may contain duplicate information. For example, the Instrument Chunk defines loop points and MIDI System Exclusive data in the MIDI Data Chunk may also define loop points. What happens if these loop points are different? How is an application supposed to loop the sound? Such conflicts are resolved by defining a precedence for chunks. This precedence is illustrated in Figure 10.

```
              Common Chunk            Highest Precedence
                   |
            Sound Data Chunk
                   |
              Marker Chunk
                   |
            Instrument Chunk
                   |
              Comment Chunk
                   |
               Name Chunk
                   |
              Author Chunk
                   |
             Copyright Chunk
                   |
             Annotation Chunk
                   |
          Audio Recording Chunk
                   |
             MIDI Data Chunk
                   |
      Application Specific Chunk   Lowest Precedence
```

Figure 10: Chunk Precedence

The Common Chunk has the highest precedence, while the Application Specific Chunk has the lowest. Information in the Common Chunk always takes precedence over conflicting information in any other chunk. The Application Specific Chunk always loses in conflicts with other chunks. By looking at the chunk hierarchy, for example, one sees that the loop points in the Instrument Chunk take precedence over conflicting loop points found in the MIDI Data Chunk.

It is the responsibility of applications that write data into the lower precedence chunks to make sure that the higher precedence chunks are updated accordingly.

Figure 11 illustrates an example of a FORM AIFF. An Audio IFF file is simple a file containing a single FORM AIFF. The FORM AIFF is stored in the data fork of Macintosh file systems that can handle resource forks.

```
 ____________________________________________________________________
| FORM AIFF                                                          |
|                                   __________                       |
|                      ckID        |_ 'FORM'  _|                      |
|                      ckSize      |_ 176516  _|                      |
|  ___________         formType    |_ 'AIFF'  _|  __________________  |
| | Common             ckID        |_ 'COMM'  _|                    | |
| | Chunk              ckSize      |_ 18      _|                    | |
| |                    numChannels |_ 2  _|____                     | |
| |               numSampleFrames  |_ 88200   _|                    | |
| |                    sampleSize  |_ 16 _|_______________________  | |
| |__________          sampleRate  |_ 44100.00 ___________________| | |
| | Marker             ckID        |_ 'MARK'  _|                    | |
| | Chunk              ckSize      |_ 34      _|                    | |
| |                    numMarkers  |_ 2  _|                         | |
| |                    id          |_ 1  _|______                   | |
| |                    position    |_ 44100    _|___________________| |
| |                    markerName  | 8 |'b'|'e'|'g'|' '|'l'|'o'|'o'|'p'| 0 | |
| |                    id          |_ 2  _|______                   | |
| |                    position    |_ 88200    _|___________________| |
| |__________          markerName  | 8 |'e'|'n'|'d'|' '|'l'|'o'|'o'|'p'| 0 | |
| | Instrument         ckID        |_ 'INST'   _|                   | |
| | Chunk              ckSize      |_ 20       _|                   | |
| |                    baseNote    | 60|                            | |
| |                    detune      | -3|                            | |
| |                    lowNote     | 57|                            | |
| |                    highNote    | 63|                            | |
| |                    lowVelocity | 1 |                            | |
| |                   highVelocity |127|__                          | |
| |                    gain        |_ 6 _|                          | |
| |   sustainLoop.playMode         |_ 1 _|                          | |
| |  sustainLoop.beginLoop         |_ 1 _|                          | |
| |    sustainLoop.endLoop         |_ 2 _|                          | |
| |   releaseLoop.playMode         |_ 0 _|                          | |
| |  releaseLoop.beginLoop         |_ - _|                          | |
| |__  releaseLoop.endLoop         |_ - _|__________________________| |
| | Sound              ckID        |_ 'SSND'  _|                    | |
| | Data               ckSize      |_ 176408  _|                    | |
| | Chunk              offset      |_ 0       _|                    | |
| |                    blockSize   |_ 0       _|                    | |
| |                    soundData   |_ch 1 _|_ch 2 _| . . . |_ch 1 _|_ch 2 _| | |
| |                    first sample frame   88200th sample frame    | |
| |_________________________________________________________________| |
|____________________________________________________________________|
```

Figure 11: Sample FORM AIFF

Further Reference

- "Inside Macintosh", Volume II, Apple Computer, Inc.
- "Apple Numerics Manual", Second Edition, Apple Computer, Inc.
- "File Type Note: File Type $D8, Auxiliary Type $0002, Apple IIGS Sampled Instrument Format", Apple Computer, Inc.
- "Audio Interchange File Format v1.3", APDA
- "AES Recommended Practice for Digital Audio Engineering--Serial Transmission Format for Linearly Represented Digital Audio Data", Audio Engineering Society, 60 East 42nd Street, New York, NY 10165
- "MIDI: Musical Instrument Digital Interface, Specification 1.0", the International MIDI Association.
- "'EA IFF 85' Standard for Interchange Format Files", Electronic Arts
- "'8SVX' IFF 8-bit Sampled Voice", Electronic Arts

```
Animated bitmap form (Framer, Deluxe Video)

TITLE:  Form ANBM (animated bitmap form used by Framer, Deluxe Video)

(note from the author)

   The format was designed for simplicity at a time when the IFF
standard was very new and strange to us all.  It was not designed
to be a general purpose animation format.  It was intended to be
a private format for use by DVideo, with the hope that a more
powerful format would emerge as the Amiga became more popular.

   I hope you will publish this format so that other formats will
not inadvertantly conflict with it.

PURPOSE:  To define simple animated bitmaps for use in DeluxeVideo.

   In Deluxe Video objects appear and move in the foreground
with a picture in the background.  Objects are "small" bitmaps
usually saved as brushes from DeluxePaint and pictures are large
full screen bitmaps saved as files from DeluxePaint.

   Two new chunk headers are defined: ANBM and FSQN.

   An animated bitmap (ANBM) is a series of bitmaps of the same
size and depth.  Each bitmap in the series is called a frame and
is labeled by a character, 'a b c ...' in the order they
appear in the file.

   The frame sequence chunk (FSQN) specifies the playback
sequence of the individual bitmaps to achieve animation.
FSQN_CYCLE and FSQN_TOFRO specify two algorithmic sequences.  If
neither of these bits is set, an arbitrary sequence can be used
instead.


    ANBM         - identifies this file as an animated bitmap
    .FSQN        - playback sequence information
    .LIST ILBM   - LIST allows following ILBMs to share properties
    ..PROP ILBM  - properties follow
    ...BMHD      - bitmap header defines common size and depth
    ...CMAP      - colormap defines common colors
    ..FORM ILBM  - first frame follows
    ..BODY       - the first frame
       .         - FORM ILBM and BODY for each remaining frame
       .
       .

Chunk Description:

   The ANBM chunk identifes this file as an animated bitmap

Chunk Spec:

   #define ANBM    MakeID('A','N','B','M')

Disk record:

   none
```

```
Chunk Description:

   The FSQN chunk specifies the frame playback sequence

Chunk Spec:

   #define FSQN    MakeID('F','S','Q','N')

   /* Flags */
   #define FSQN_CYCLE  0x0001  /* Ignore sequence, cycle a,b,..y,z,a,b,.. */
   #define FSQN_TOFRO  0x0002  /* Ignore sequence, cycle a,b,..y,z,y,..a,b, */
   /* Disk record */
   typedef struct {
       WORD numframes;      /* Number of frames in the sequence */
       LONG dt;             /* Nominal time between frames in jiffies */
       WORDBITS flags;      /* Bits modify behavior of the animation */
       UBYTE sequence[80];  /* string of 'a'..'z' specifying sequence */
       } FrameSeqn;


Supporting Software:

   DeluxeVideo by Mike Posehn and Tom Case for Electronic Arts


Thanks,
   Mike Posehn
```

```
ANIM brush format

                    Dpaint Anim Brush IFF Format

             From a description by the author of DPaint,
                      Dan Silva, Electronic Arts


The "Anim Brushes" of DPaint III are saved on disk in the IFF "ANIM" format.
Basically, an ANIM Form consists of an initial ILBM which is the first frame
of the animation, and any number of subsequent "ILBM"S (which aren't really
ILBM's) each of which contains an ANHD animation header chunk and a DLTA chunk
comprised of the encoded difference between a frame and a previous one.

To use ANIM terminology (for a description of the ANIM format, see the IFF
Anim Spec, by Gary Bonham). Anim Brushes use a "type 5" encoding, which is
a vertical, byte-oriented delta encoding (based on Jim Kent's RIFF).  The
deltas have an interleave of 1, meaning deltas are computed between adjacent
frames, rather than between frames 2 apart, which is the usual ANIM custom
for the purpose of fast hardware page-flipping.  Also, the deltas use
Exclusive Or to allow reversable play.

However, to my knowledge, all the existing Anim players in the Amiga world
will only play type 5 "Anim"s which have an interleave of 0 (i.e. 2) and
which use a Store operation rather than Exclusive Or, so no existing programs
will read Anim Brushes anyway.  The job of modifying existing Anim readers
to read Anim Brushes should be simplified, however.


Here is an outline of the structure of the IFF Form output by DPaint III as
an "Anim Brush".  The IFF Reader should of course be flexible enough to
tolerate variation in what chunks actually appear in the initial ILBM.

                  FORM ANIM
                      . FORM ILBM         first frame
                      . . BMHD
                      . . CMAP
                      . . DPPS
                      . . GRAB
                      . . CRNG
                      . . CRNG
                      . . CRNG
                      . . CRNG
                      . . CRNG
                      . . CRNG
                      . . DPAN     my own little chunk.
                      . . CAMG
                      . . BODY

                      . FORM ILBM         frame 2
                      . . ANHD                animation header chunk
                      . . DLTA                delta mode data

                      . FORM ILBM         frame 3
                      . . ANHD                animation header chunk
                      . . DLTA                delta mode data

                      . FORM ILBM         frame 4
                      . . ANHD                animation header chunk
                      . . DLTA                delta mode data
         ...
                   . FORM ILBM         frame N
                      . . ANHD                animation header chunk
                      . . DLTA                delta mode data
```

```
--- Here is the format of the DPAN chunk:

typedef struct {
 UWORD version;   /* current version=4 */
 UWORD nframes;   /* number of frames in the animation.*/
 ULONG flags;   /* Not used */
 } DPAnimChunk;

The version number was necessary during development. At present all I look
at is "nframes".


--- Here is the ANHD chunk format:

typedef struct {
 UBYTE operation;  /* =0  set directly
        =1  XOR ILBM mode,
        =2 Long Delta mode,
        =3 Short Delta mode
        =4 Generalize short/long Delta mode,
        =5 Byte Vertical Delta (riff)
        =74 (Eric Grahams compression mode)
    */
 UBYTE mask;      /* XOR ILBM only: plane mask where data is*/
 UWORD w,h;
 WORD x,y;
 ULONG abstime;
 ULONG reltime;
 UBYTE interleave; /* 0 defaults to 2 */
 UBYTE pad0;   /* not used */
 ULONG bits;   /* meaning of bits:
    bit# =0                  =1
    0    short data          long data
    1    store               XOR
    2    separate info       one info for
         for each plane      for all planes
    3    not RLC             RLC (run length encoded)
    4    horizontal          vertical
    5    short info offsets long info offsets
   -------------------------*/
 UBYTE pad[16];
 } AnimHdr;


for Anim Brushes, I set:

 animHdr.operation = 5;  /* RIFF encoding */
 animHdr.interleave = 1;
 animHdr.w = curAnimBr.bmob.pict.box.w;
 animHdr.h = curAnimBr.bmob.pict.box.h;
 animHdr.reltime = 1;
 animHdr.abstime = 0;
 animHdr.bits = 4; /* indicating XOR */

-- everything else is set to 0.

NOTE: the "bits" field was actually intended ( by the original creator of
the ANIM format, Gary Bonham of SPARTA, Inc.) for use with only with
compression method 4. I am using bit 2 of the bits field to indicate the
Exclusive OR operation in the context of method 5, which seems like a
reasonable generalization.
```

For an Anim Brush with 10 frames, there will be an initial frame followed by 10 Delta's (i.e ILBMS containing ANHD and DLTA chunks). Applying the first Delta to the initial frame generates the second frame, applying the second Delta to the second frame generates the third frame, etc. Applying the last Delta thus brings back the first frame.

The DLTA chunk begins with 16 LONG plane offets, of which DPaint only uses the first 6 (at most). These plane offsets are either the offset (in bytes) from the beginning of the DLTA chunk to the data for the corresponding plane, or Zero, if there was no change in that plane. Thus the first plane offset is either 0 or 64.

(The following description of the method is based on Gary Bonham's rewording of Jim Kent's RIFF documentation.)

Compression/decompression is performed on a plane-by-plane basis.

Each byte-column of the bitplane is compressed separately. A 320x200 bitplane would have 40 columns of 200 bytes each. In general, the bitplanes are always an even number of bytes wide, so for instance a 17x20 bitplane would have 4 columns of 20 bytes each.

Each column starts with an op-count followed by a number of ops. If the op-count is zero, that's ok, it just means there's no change in this column from the last frame. The ops are of three kinds, and followed by a varying amount of data depending on which kind:

1. SKIP - this is a byte with the hi bit clear that says how many rows to move the "dest" pointer forward, ie to skip. It is non-zero.

2. DUMP - this is a byte with the hi bit set. The hi bit is masked off and the remainder is a count of the number of bytes of data to XOR directly. It is followed by the bytes to copy.

3. RUN - this is a 0 byte followed by a count byte, followed by a byte value to repeat "count" times, XOR'ing it into the destination.

Bear in mind that the data is compressed vertically rather than horizontally, so to get to the next byte in the destination you add the number of bytes per row instead of one.

The Format of DLTA chunks is as described in section 2.2.2 of the Anim Spec. The encoding for type 5 is described in section 2.2.3 of the Anim Spec.

Cel animation form

A N I M
An IFF Format For CEL Animations

Revision date: 4 May 1988

prepared by:
SPARTA Inc.
23041 de la Carlota
Laguna Hills, Calif 92653
(714) 768-8161
contact: Gary Bonham

also by:
Aegis Development Co.
2115 Pico Blvd.
Santa Monica, Calif 90405
213) 392-9972

1.0 Introduction

The ANIM IFF format was developed at Sparta originally for the production of animated video sequences on the Amiga computer. The intent was to be able to store, and play back, sequences of frames and to minimize both the storage space on disk (through compression) and playback time (through efficient de-compression algorithms). It was desired to maintain maximum compatibility with existing IFF formats and to be able to display the initial frame as a normal still IFF picture.

Several compression schemes have been introduced in the ANIM format. Most of these are strictly of historical interest as the only one currently being placed in new code is the vertical run length encoded byte encoding developed by Jim Kent.

1.1 ANIM Format Overview

The general philosophy of ANIMs is to present the initial frame as a normal, run-length-encoded, IFF picture. Subsequent frames are then described by listing only their differences from a previous frame. Normally, the "previous" frame is two frames back as that is the frame remaining in the hidden screen buffer when double-buffering is used. To better understand this, suppose one has two screens, called A and B, and the ability to instantly switch the display from one to the other. The normal playback mode is to load the initial frame into A and duplicate it into B. Then frame A is displayed on the screen. Then the differences for frame 2 are used to alter screen B and it is displayed. Then the differences for frame 3 are used to alter screen A and it is displayed, and so on. Note that frame 2 is stored as differences from frame 1, but all other frames are stored as differences from two frames back.

ANIM is an IFF FORM and its basic format is as follows (this assumes the reader has a basic understanding of IFF format files):

```
                FORM ANIM
                . FORM ILBM         first frame
                . . BMHD                normal type IFF data
                . . ANHD                optional animation header
                                        chunk for timing of 1st frame.
                . . CMAP
                . . BODY
                . FORM ILBM         frame 2
                . . ANHD                animation header chunk
                . . DLTA                delta mode data
                . FORM ILBM         frame 3
                . . ANHD
                . . DLTA
                     ...
```

The initial FORM ILBM can contain all the normal ILBM chunks, such as CRNG, etc. The BODY will normally be a standard run-length-encoded data chunk (but may be any other legal compression mode as indicated by the BMHD). If desired, an ANHD chunk can appear here to provide timing data for the first frame. If it is here, the operation field should be =0.

The subsequent FORMs ILBM contain an ANHD, instead of a BMHD, which duplicates some of BMHD and has additional parameters pertaining to the animation frame. The DLTA chunk contains the data for the delta compression modes. If the older XOR compression mode is used, then a BODY chunk will be here. In addition, other chunks may be placed in each of these as deemed necessary (and as code is placed in player programs to utilize them). A good example would be CMAP chunks to alter the color palette. A basic assumption in ANIMs is that the size of the bitmap, and the display mode (e.g. HAM) will not change through the animation. Take care when playing an ANIM that if a CMAP occurs with a frame, then the change must be applied to both buffers.

Note that the DLTA chunks are not interleaved bitmap representations, thus the use of the ILBM form is inappropriate for these frames. However, this inconsistency was not noted until there were a number of commercial products either released or close to release which generated/played this format. Therefore, this is probably an inconsistency which will have to stay with us.

1.2 Recording ANIMs

To record an ANIM will require three bitmaps - one for creation of the next frame, and two more for a "history" of the previous two frames for performing the compression calculations (e.g. the delta mode calculations).

There are five frame-to-frame compression methods currently defined. The first three are mainly for historical interest. The product Aegis VideoScape 3D utilizes the third method in version 1.0, but switched to method 5 on 2.0. This is the only instance known of a commercial product generating ANIMs of any of the first three methods. The fourth method is a general short or long word compression scheme which has several options including whether the compression is horizontal or vertical, and whether or not it is XOR format. This offers a choice to the user for the optimization of file size and/or playback speed. The fifth method is the byte vertical run length encoding as designed by Jim Kent. Do not confuse this with Jim's RIFF file format which is different than ANIM. Here we utilized his compression/decompression routines within the ANIM file structure.

The following paragraphs give a general outline of each of the methods of compression currently included in this spec.

1.2.1 XOR mode

This mode is the original and is included here for historical interest. In general, the delta modes are far superior. The creation of XOR mode is quite simple. One simply performs an exclusive-or (XOR) between all corresponding bytes of the new frame and two frames back. This results in a new bitmap with 0 bits wherever the two frames were identical, and 1 bits where they are different. Then this new bitmap is saved using run-length-encoding. A major obstacle of this mode is in the time consumed in performing the XOR upon reconstructing the image.

1.2.2 Long Delta mode

This mode stores the actual new frame long-words which are different, along with the offset in the bitmap. The exact format is shown and discussed in section 2 below. Each plane is handled separately, with no data being saved if no changes take place in a given plane. Strings of 2 or more long-words in a row which change can be run together so offsets do not have to be saved for each one.

Constructing this data chunk usually consists of having a buffer to hold the data, and calculating the data as one compares the new frame, long-word by long-word, with two frames back.

1.2.3 Short Delta mode

This mode is identical to the Long Delta mode except that short-words are saved instead of long-words. In most instances, this mode results in a smaller DLTA chunk. The Long Delta mode is mainly of interest in improving the playback speed when used on a 32-bit 68020 Turbo Amiga.

1.2.4 General Delta mode

The above two delta compression modes were hastily put together. This mode was an attempt to provide a well-thought-out delta compression scheme. Options provide for both short and long word compression, either vertical or horizontal compression, XOR mode (which permits reverse playback), etc. About the time this was being finalized, the fifth mode, below, was developed by Jim Kent. In practice the short-vertical-run-length-encoded deltas in this mode play back faster than the fifth mode (which is in essence a byte-vertical-run-length-encoded delta mode) but does not compress as well - especially for very noisy data such as digitized images. In most cases, playback speed not being terrifically slower, the better compression (sometimes 2x) is preferable due to limited storage media in most machines.

Details on this method are contained in section 2.2.2 below.

1.2.5 Byte Vertical Compression

This method does not offer the many options that method 4 offers, but is very successful at producing decent compression even for very noisy data such as digitized images. The method was devised by Jim Kent and is utilized in his RIFF file format which is different than the ANIM format. The description of this method in this document is taken from Jim's writings. Further, he has released both compression and decompression code to public domain.

```
    Details on this method are contained in section 2.2.3 below.

1.3 Playing ANIMs

  Playback of ANIMs will usually require two buffers, as mentioned
  above, and double-buffering between them.  The frame data from
  the ANIM file is used to modify the hidden frame to the next
  frame to be shown.  When using the XOR mode, the usual run-
  length-decoding routine can be easily modified to do the
  exclusive-or operation required.  Note that runs of zero bytes,
  which will be very common, can be ignored, as an exclusive or
  of any byte value to a byte of zero will not alter the original
  byte value.

  The general procedure, for all compression techniques, is to first
  decode the initial ILBM picture into the hidden buffer and double-
  buffer it into view.  Then this picture is copied to the other (now
  hidden) buffer.  At this point each frame is displayed with the
  same procedure.  The next frame is formed in the hidden buffer by
  applying the DLTA data (or the XOR data from the BODY chunk in the
  case of the first XOR method) and the new frame is double-buffered
  into view.  This process continues to the end of the file.

  A master colormap should be kept for the entire ANIM which would
  be initially set from the CMAP chunk in the initial ILBM.  This
  colormap should be used for each frame.  If a CMAP chunk appears
  in one of the frames, then this master colormap is updated and the
  new colormap applies to all frames until the occurrance of another
  CMAP chunk.

  Looping ANIMs may be constructed by simply making the last two frames
  identical to the first two.  Since the first two frames are special
  cases (the first being a normal ILBM and the second being a delta from
  the first) one can continually loop the anim by repeating from frame
  three.  In this case the delta for creating frame three will modify
  the next to the last frame which is in the hidden buffer (which is
  identical to the first frame), and the delta for creating frame four
  will modify the last frame which is identical to the second frame.

  Multi-File ANIMs are also supported so long as the first two frames
  of a subsequent file are identical to the last two frames of the
  preceeding file.  Upon reading subsequent files, the ILBMs for the
  first two frames are simply ignored, and the remaining frames are
  simply appended to the preceeding frames.  This permits splitting
  ANIMs across multiple floppies and also permits playing each section
  independently and/or editing it independent of the rest of the ANIM.

  Timing of ANIM playback is easily achieved using the vertical blank
  interrupt of the Amiga.  There is an example of setting up such
  a timer in the ROM Kernel Manual.  Be sure to remember the timer
  value when a frame is flipped up, so the next frame can be flipped
  up relative to that time.  This will make the playback independent
  of how long it takes to decompress a frame (so long as there is enough
  time between frames to accomplish this decompression).
```

```
2.0 Chunk Formats
   2.1 ANHD Chunk
      The ANHD chunk consists of the following data structure:

           UBYTE operation  The compression method:
                            =0 set directly (normal ILBM BODY),
                            =1 XOR ILBM mode,
                            =2 Long Delta mode,
                            =3 Short Delta mode,
                            =4 Generalized short/long Delta mode,
                            =5 Byte Vertical Delta mode
                            =6 Stereo op 5 (third party)
                            =74 (ascii 'J') reserved for Eric Graham's
                               compression technique (details to be
                               released later).

           UBYTE mask      (XOR mode only - plane mask where each
                            bit is set =1 if there is data and =0
                            if not.)
           UWORD w,h       (XOR mode only - width and height of the
                            area represented by the BODY to eliminate
                            unnecessary un-changed data)
           WORD  x,y       (XOR mode only - position of rectangular
                            area representd by the BODY)
           ULONG abstime   (currently unused - timing for a frame
                            relative to the time the first frame
                            was displayed - in jiffies (1/60 sec))
           ULONG reltime   (timing for frame relative to time
                            previous frame was displayed - in
                            jiffies (1/60 sec))
           UBYTE interleave (unused so far - indicates how may frames
                             back this data is to modify.  =0 defaults
                             to indicate two frames back (for double
                             buffering). =n indicates n frames back.
                             The main intent here is to allow values
                             of =1 for special applications where
                             frame data would modify the immediately
                             previous frame)
           UBYTE pad0        Pad byte, not used at present.
           ULONG bits        32 option bits used by options=4 and 5.
                             At present only 6 are identified, but the
                             rest are set =0 so they can be used to
                             implement future ideas.  These are defined
                             for option 4 only at this point.  It is
                             recommended that all bits be set =0 for
                             option 5 and that any bit settings used in
                             the future (such as for XOR mode) be compatible
                             with the option 4 bit settings.   Player code
                             should check undefined bits in options 4 and 5
                             to assure they are zero.

                             The six bits for current use are:

                             bit #              set =0               set =1
                             ===============================================
                             0               short data           long data
                             1                  set                  XOR
                             2              separate info        one info list
                                            for each plane       for all planes
                             3                not RLC         RLC (run length coded)
                             4               horizontal           vertical
                             5             short info offsets   long info offsets

           UBYTE pad[16]     This is a pad for future use for future
                             compression modes.
```

2.2 DLTA Chunk

This chunk is the basic data chunk used to hold delta compression data. The format of the data will be dependent upon the exact compression format selected. At present there are two basic formats for the overall structure of this chunk.

2.2.1 Format for methods 2 & 3

This chunk is a basic data chunk used to hold the delta compression data. The minimum size of this chunk is 32 bytes as the first 8 long-words are byte pointers into the chunk for the data for each of up to 8 bitplanes. The pointer for the plane data starting immediately following these 8 pointers will have a value of 32 as the data starts in the 33-rd byte of the chunk (index value of 32 due to zero-base indexing).

The data for a given plane consists of groups of data words. In Long Delta mode, these groups consist of both short and long words - short words for offsets and numbers, and long words for the actual data. In Short Delta mode, the groups are identical except data words are also shorts so all data is short words. Each group consists of a starting word which is an offset. If the offset is positive then it indicates the increment in long or short words (whichever is appropriate) through the bitplane. In other words, if you were reconstructing the plane, you would start a pointer (to shorts or longs depending on the mode) to point to the first word of the bitplane. Then the offset would be added to it and the following data word would be placed at that position. Then the next offset would be added to the pointer and the following data word would be placed at that position. And so on... The data terminates with an offset equal to 0xFFFF.

A second interpretation is given if the offset is negative. In that case, the absolute value is the offset+2. Then the following short-word indicates the number of data words that follow. Following that is the indicated number of contiguous data words (longs or shorts depending on mode) which are to be placed in contiguous locations of the bitplane.

If there are no changed words in a given plane, then the pointer in the first 32 bytes of the chunk is =0.

2.2.2 Format for method 4

The DLTA chunk is modified slightly to have 16 long pointers at the start. The first 8 are as before - pointers to the start of the data for each of the bitplanes (up to a theoretical max of 8 planes). The next 8 are pointers to the start of the offset/numbers data list. If there is only one list of offset/numbers for all planes, then the pointer to that list is repeated in all positions so the playback code need not even be aware of it. In fact, one could get fancy and have some bitplanes share lists while others have different lists, or no lists (the problems in these schemes lie in the generation, not in the playback).

The best way to show the use of this format is in a sample playback routine.

```
SetDLTAshort(bm,deltaword)
struct BitMap *bm;
WORD *deltaword;
{
   int i;
   LONG *deltadata;
   WORD *ptr,*planeptr;
   register int s,size,nw;
   register WORD *data,*dest;

   deltadata = (LONG *)deltaword;
   nw = bm->BytesPerRow >>1;

   for (i=0;i<bm->Depth;i++) {
      planeptr = (WORD *)(bm->Planes[i]);
      data = deltaword + deltadata[i];
      ptr  = deltaword + deltadata[i+8];
      while (*ptr != 0xFFFF) {
         dest = planeptr + *ptr++;
         size = *ptr++;
         if (size < 0) {
            for (s=size;s<0;s++) {
               *dest = *data;
               dest += nw;
            }
            data++;
         }
         else {
            for (s=0;s<size;s++) {
               *dest = *data++;
               dest += nw;
            }
         }
      }
   }
   return(0);
}
```

The above routine is for short word vertical compression with run length compression. The most efficient way to support the various options is to replicate this routine and make alterations for, say, long word or XOR. The variable nw indicates the number of words to skip to go down the vertical column. This one routine could easily handle horizontal compression by simply setting nw=1. For ultimate playback speed, the core, at least, of this routine should be coded in assembly language.

2.2.2 Format for method 5

In this method the same 16 pointers are used as in option 4. The first 8 are pointers to the data for up to 8 planes. The second set of 8 are not used but were retained for several reasons. First to be somewhat compatible with code for option 4 (although this has not proven to be of any benefit) and second, to allow extending the format for more bitplanes (code has been written for up to 12 planes).

Compression/decompression is performed on a plane-by-plane basis. For each plane, compression can be handled by the skip.c code (provided Public Domain by Jim Kent) and decompression can be handled by unvscomp.asm (also provided Public Domain by Jim Kent).

Compression/decompression is performed on a plane-by-plane basis. The following description of the method is taken directly from Jim Kent's code with minor re-wording. Please refer to Jim's code (skip.c and unvscomp.asm) for more details:

```
    Each column of the bitplane is compressed separately.
    A 320x200 bitplane would have 40 columns of 200 bytes each.
    Each column starts with an op-count followed by a number
    of ops.  If the op-count is zero, that's ok, it just means
    there's no change in this column from the last frame.
    The ops are of three classes, and followed by a varying
    amount of data depending on which class:
      1. Skip ops - this is a byte with the hi bit clear that
         says how many rows to move the "dest" pointer forward,
         ie to skip. It is non-zero.
      2. Uniq ops - this is a byte with the hi bit set.  The hi
         bit is masked down and the remainder is a count of the
         number of bytes of data to copy literally.  It's of
         course followed by the data to copy.
      3. Same ops - this is a 0 byte followed by a count byte,
         followed by a byte value to repeat count times.
    Do bear in mind that the data is compressed vertically rather
    than horizontally, so to get to the next byte in the destination
    we add the number of bytes per row instead of one!
```

2-D Object standard format

FORM DR2D

Description by Ross Cunniff and John Orr

A standard IFF FORM to describe 2D drawings has been sorely needed for a long time. Several commercial drawing packages have been available for some time but none has established its file format as the Amiga standard. The absence of a 2D drawing standard hinders the development of applications that use 2D drawings as it forces each application to understand several private standards instead of a single one. Without a standard, data exchange for both the developer and the user is difficult, if not impossible.

The DR2D FORM fills this void. This FORM was developed by Taliesin, Inc. for use as the native file format for their two-dimensional structured drawing package, ProVector. Saxon Industries and Soft Logik Publishing Corporation are planning to support this new FORM in the near future.

Many of the values stored in the DR2D FORM are stored as IEEE single precision floating point numbers. These numbers consist of 32 bits, arranged as follows:

```
 _______________________________________________________________________
| s e e e e e e e | e m m m m m m m | m m m m m m m m | m m m m m m m m |
 -----------------------------------------------------------------------
  31            24  23            16  15             8   7             0
```

where:

```
        s       is the sign of the number where 1 is negative and 0 is
                  positive.
        e       is the 8 bit exponent in excess 127 form.  This number
                  is the power of two to which the mantissa is raised
                  (Excess 127 form means that 127 is added to the
                  exponent before packing it into the IEEE number.)
        m       is the 23 bit mantissa.  It ranges from 1.0000000 to
                  1.999999..., where the leading base-ten one is
                  assumed.
```

An IEEE single precision with the value of 0.0000000 has all its bits cleared.

The DR2D Chunks

```
FORM (0x464F524D)       /* All drawings are a FORM */

        struct FORMstruct {
            ULONG       ID;             /* DR2D */
            ULONG       Size;
        };

DR2D (0x44523244)  /* ID of 2D drawing */
```

The DR2D chunks are broken up into three groups: the global drawing attribute chunks, the object attribute chunks, and the object chunks. The global drawing attribute chunks describe elements of a 2D drawing that are common to many objects in the drawing. Document preferences, palette information, and custom fill patterns are typical document-wide settings defined in global drawing attribute chunks. The object attribute chunks are used to set certain properties of the object chunk(s) that follows the object attribute chunk. The current fill pattern, dash pattern, and line color are all set using an object attribute chunk. Object chunks describe the actual DR2D drawing. Polygons, text, and bitmaps are found in these chunks.

The Global Drawing Attribute Chunks

The following chunks describe global attributes of a DR2D document.

```
DRHD (0x44524844)       /* Drawing header */
```

The DRHD chunk contains the upper left and lower right extremes of the document in (X, Y) coordinates. This chunk is required and should only appear once in a document in the outermost layer of the DR2D file (DR2Ds can be nested).

```
        struct DRHDstruct {
            ULONG       ID;
            ULONG       Size;                   /* Always 16 */
            IEEE        XLeft, YTop,
                        XRight, YBot;
        };
```

The point (XLeft,YTop) is the upper left corner of the project and the point (XRight,YBot) is its lower right corner. These coordinates not only supply the size and position of the document in a coordinate system, they also supply the project's orientation. If XLeft < XRight, the X-axis increases toward the right. If YTop < YBot, the Y-axis increases toward the bottom. Other combinations are possible; for example in Cartesian coordinates, XLeft would be less than XRight but YTop would be greater than YBot.

```
PPRF (0x50505249)       /* Page preferences */
```

The PPRF chunk contains preference settings for ProVector. Although this chunk is not required, its use is encouraged because it contains some important environment information.

```
        struct PPRFstruct {
            ULONG       ID;
            ULONG       Size;
            char        Prefs[Size];
        };
```

DR2D stores preferences as a concatenation of several null-terminated strings, in the Prefs[] array. The strings can appear in any order. The currently supported strings are:

```
            Units=<unit-type>
            Portrait=<boolean>
            PageType=<page-type>
            GridSize=<number>

where:
            <unit-type>     is either Inch, Cm, or Pica
            <boolean>       is either True or False
            <page-type>     is either Standard, Legal, B4, B5, A3,
                              A4, A5, or Custom
            <number>        is a floating-point number
```

The DR2D FORM does not require this chunk to explicitly state all the possible preferences. In the absence of any particular preference string, a DR2D reader should fall back on the default value. The defaults are:

```
            Units=Inch
            Portrait=True
            PageType=Standard
            GridSize=1.0
```

```
CMAP (0x434D4150)       /* Color map (Same as ILBM CMAP) */
```

This chunk is identical to the ILBM CMAP chunk as described in the IFF ILBM documentation.

```
        struct CMAPstruct {
            ULONG       ID;
            ULONG       Size;
            UBYTE       ColorMap[Size];
        };
```

ColorMap is an array of 24-bit RGB color values. The 24-bit value is spread across three bytes, the first of which contains the red intensity, the next contains the green intensity, and the third contains the blue intensity. Because DR2D stores its colors with 24-bit accuracy, DR2D readers must not make the mistake that some ILBM readers do in assuming the CMAP chunk colors correspond directly to Amiga color registers.

```
FONS (0x464F4E53)       /* Font chunk (Same as FTXT FONS chunk) */
```

The FONS chunk contains information about a font used in the DR2D FORM. ProVector does not include support for Amiga fonts. Instead, ProVector uses fonts defined in the OFNT FORM which is documented later in this article.

```
        struct FONSstruct {
            ULONG       ID;
            ULONG       Size;
            UBYTE       FontID;         /* ID the font is referenced by */
            UBYTE       Pad1;           /* Always 0 */
            UBYTE       Proportional;   /* Is it proportional? */
            UBYTE       Serif;          /* does it have serifs? */
            CHAR        Name[Size-4];   /* The name of the font */
        };
```

The UBYTE FontID field is the number DR2D assigns to this font. References to this font by other DR2D chunks are made using this number.

The Proportional and Serif fields indicate properties of this font. Specifically, Proportional indicates if this font is proportional, and Serif indicates if this font has serifs. These two options were created to allow for font substitution in case the specified font is not available. They are set according to these values:

0	The DR2D writer didn't know if this font is proportional/has serifs.
1	No, this font is not proportional/does not have serifs.
2	Yes, this font is proportional/does have serifs.

The last field, Name[], is a NULL terminated string containing the name of the font.

```
DASH (0x44415348)       /* Line dash pattern for edges */
```

This chunk describes the on-off dash pattern associated with a line.

```
        struct DASHstruct {
            ULONG       ID;
            ULONG       Size;
            USHORT      DashID;                 /* ID of the dash pattern */
            USHORT      NumDashes;              /* Should always be even */
            IEEE        Dashes[NumDashes];      /* On-off pattern */
        };
```

DashID is the number assigned to this specific dash pattern. References to this dash pattern by other DR2D chunks are made using this number.

The Dashes[] array contains the actual dash pattern. The first number in the array (element 0) is the length of the ''on'' portion of the pattern. The second number (element 1) specifies the ''off'' portion of the pattern. If there are more entries in the Dashes array, the pattern will continue. Even-index elements specify the length of an ''on'' span, while odd-index elements specify the length of an ''off'' span. There must be an even number of entries. These lengths are not in the same units as specified in the PPRF chunk, but are multiples of the line width, so a line of width 2.5 and a dash pattern of 1.0, 2.0 would have an ''on'' span of length 1.0 x 2.5 = 2.5 followed by an ''off'' span of length 2.0 x 2.5 = 5. The following figure shows several dash pattern examples. Notice that for lines longer than the dash pattern, the pattern repeats.

[figure 1 - dash patterns]

By convention, DashID 0 is reserved to mean 'No line pattern at all', i.e. the edges are invisible. This DASH pattern should not be defined by a DR2D DASH chunk. Again by convention, a NumDashes of 0 means that the line is solid.

```
AROW (0x41524F57)       /* An arrow-head pattern */
```

The AROW chunk describes an arrowhead pattern. DR2D open polygons (OPLY) can have arrowheads attached to their endpoints. See the description of the OPLY chunk later in this article for more information on the OPLY chunk.

```
        #define ARROW_FIRST  0x01 /* Draw an arrow on the OPLY's first point */
        #define ARROW_LAST   0x02 /* Draw an arrow on the OPLY's last point */

        struct AROWstruct {
            ULONG       ID;
            ULONG       Size;
            UBYTE       Flags;          /* Flags, from ARROW_*, above */
            UBYTE       Pad0;           /* Should always 0 */
            USHORT      ArrowID;        /* Name of the arrow head */
            USHORT      NumPoints;
            IEEE        ArrowPoints[NumPoints*2];
        };
```

The Flags field specifies which end(s) of an OPLY to place an arrowhead based on the #defines above. ArrowID is the number by which an OPLY will reference this arrowhead pattern.

The coordinates in the array ArrowPoints[] define the arrowhead's shape. These points form a closed polygon. See the section on the OPLY/CPLY object chunks for a descriptionof how DR2D defines shapes. The arrowhead is drawn in the same coordinate system relative to the endpoint of the OPLY the arrowhead is attached to. The arrowhead's origin (0,0) coincides with the OPLY's endpoint. DR2D assumes that the arrowhead represented in the AROW chunk is pointing to the right so the proper rotation can be applied to the arrowhead. The arrow is filled according to the current fill pattern set in the ATTR object attribute chunk.

```
FILL (0x46494C4C)       /* Object-oriented fill pattern */
```

The FILL chunk defines a fill pattern. This chunk is only valid inside nested DR2D FORMs. The GRUP object chunk section of this article contans an example of the FILL chunk.

```
        struct FILLstruct {
            ULONG       ID;
            ULONG       Size;
            USHORT      FillID;                 /* ID of the fill */
        };
```

FillID is the number by which the ATTR object attribute chunk references fill patterns. The FILL chunk must be the first chunk inside a nested DR2D FORM. A FILL is followed by one DR2D object plus any of the object attribute chunks (ATTR, BBOX) associated with the object.

[Figure 2 - fill patterns]

DR2D makes a ''tile'' out of the fill pattern, giving it a virtual bounding box based on the extreme X and Y values of the FILL's object (Fig. A). The bounding box shown in Fig. A surrounding the pattern (the two ellipses) is invisible to the user. In concept, this rectangle is pasted on the page left to right, top to bottom like floor tiles (Fig. B). Again, the bounding boxes are not visible. The only portion of this tiled pattern that is visible is the part that overlaps the object (Fig. C) being filled. The object's path is called a clipping path, as it ''clips'' its shape from the tiled pattern (Fig. D). Note that the fill is only masked on top of underlying objects, so any ''holes'' in the pattern will act as a window, leaving visible underlying objects.

```
LAYR (0x4C415952)       /* Define a layer */
```

A DR2D project is broken up into one or more layers. Each DR2D object is in one of these layers. Layers provide several useful features. Any particular layer can be ''turned off'', so that the objects in the layer are not displayed. This eliminates the unnecessary display of objects not currently needed on the screen. Also, the user can lock a layer to protect the layer's objects from accidental changes.

```
        struct LAYRstruct {
            ULONG    ID;
            ULONG    Size;
            USHORT   LayerID;              /* ID of the layer */
            char     LayerName[16];        /* Null terminated and padded */
            UBYTE    Flags;                /* Flags, from LF_*, below */
            UBYTE    Pad0;                 /* Always 0 */
        };
```

LayerID is the number assigned to this layer. As the field's name indicates, LayerName[] is the NULL terminated name of the layer. Flags is a bit field who's bits are set according to the #defines below:

```
        #define LF_ACTIVE       0x01    /* Active for editing */
        #define LF_DISPLAYED    0x02    /* Displayed on the screen */
```

If the LF_ACTIVE bit is set, this layer is unlocked. A set LF_DISPLAYED bit indicates that this layer is currently visible on the screen. A cleared LF_DISPLAYED bit implies that LF_ACTIVE is not set. The reason for this is to keep the user from accidentally editing layers that are invisible.

The Object Attribute Chunks

```
ATTR (0x41545452)       /* Object attributes */
```

The ATTR chunk sets various attributes for the objects that follow it. The attributes stay in effect until the next ATTR changes the attributes, or the enclosing FORM ends, whichever comes first.

```
        /* Various fill types */
        #define FT_NONE         0    /* No fill                   */
        #define FT_COLOR        1    /* Fill with color from palette */
        #define FT_OBJECTS      2    /* Fill with tiled objects */


        struct ATTRstruct {
            ULONG       ID;
            ULONG       Size;
            UBYTE       FillType;    /* One of FT_*, above      */
            UBYTE       JoinType;    /* One of JT_*, below      */
            UBYTE       DashPattern; /* ID of edge dash pattern */
            UBYTE       ArrowHead;   /* ID of arrowhead to use  */
            USHORT      FillValue;   /* Color or object with which to fill */
            USHORT      EdgeValue;   /* Edge color index        */
            USHORT      WhichLayer;  /* ID of layer it's in     */
            IEEE        EdgeThick;   /* Line width              */
        };
```

FillType specifies what kind of fill to use on this ATTR chunk's objects. A value of FT_NONE means that this ATTR chunk's objects are not filled. FT_COLOR indicates that the objects should be filled in with a color. That color's ID (from the CMAP chunk) is stored in the FillValue field. If FillType is equal to FT_OBJECTS, FillValue contains the ID of a fill pattern defined in a FILL chunk.

JoinType determines which style of line join to use when connecting the edges of line segments. The field contains one of these four values:

```
        /* Join types */
        #define JT_NONE    0          /* Don't do line joins */
        #define JT_MITER        1     /* Mitered join */
        #define JT_BEVEL        2     /* Beveled join */
        #define JT_ROUND        3     /* Round join */
```

DashPattern and ArrowHead contain the ID of the dash pattern and arrow head for this ATTR's objects. A DashPattern of zero means that there is no dash pattern so lines will be invisible. If ArrowHead is 0, OPLYs have no arrow head. EdgeValue is the color of the line segments. WhichLayer contains the ID of the layer this ATTR's objects are in. EdgeThick is the width of this ATTR's line segments.

```
BBOX (0x42424F48)       /* Bounding box of next object in FORM */
```

The BBOX chunk supplies the dimensions and position of a bounding box surrounding the DR2D object that follows this chunk in the FORM. A BBOX chunk can apply to a FILL or AROW as well as a DR2D object. The BBOX chunk appears just before its DR2D object, FILL, or AROW chunk.

```
        struct BBOXstruct {
            ULONG       ID;
            ULONG       Size;
            IEEE                XMin, YMin,     /* Bounding box of obj. */
                                XMax, YMax;     /* including line width */
        };
```

In a Cartesian coordinate system, the point (XMin, YMin) is the coordinate of the lower left hand corner of the bounding box and (XMax, YMax) is the upper right. These coordinates take into consideration the width of the lines making up the bounding box.

```
XTRN (0x5854524E)       /* Externally controlled object */
```

The XTRN chunk was created primarily to allow ProVector to link DR2D objects to ARexx functions.

```
        struct XTRNstruct {
            ULONG   ID;
            ULONG   Size;
            short   ApplCallBacks;            /* From #defines, below */
            short   ApplNameLength;
            char    ApplName[ApplNameLength];  /* Name of ARexx func to call */
        };
```

ApplName[] contains the name of the ARexx script ProVector calls when the user manipulates the object in some way. The ApplCallBacks field specifies the particular action that triggers calling the ARexx script according to the #defines listed below.

```
        /* Flags for ARexx script callbacks */
        #define    X_CLONE      0x0001    /* The object has been cloned */
        #define    X_MOVE       0x0002    /* The object has been moved */
        #define    X_ROTATE     0x0004    /* The object has been rotated */
        #define    X_RESIZE     0x0008    /* The object has been resized */
        #define    X_CHANGE     0x0010    /* An attribute (see ATTR) of the
                                             object has changed */
        #define    X_DELETE     0x0020    /* The object has been deleted */
        #define    X_CUT        0x0040    /* The object has been deleted, but
                                             stored in the clipboard */
        #define    X_COPY       0x0080    /* The object has been copied to the
                                             clipboard */
        #define    X_UNGROUP    0x0100    /* The object has been ungrouped */
```

For example, given the XTRN object:

```
        FORM xxxx DR2D {
                XTRN xxxx { X_RESIZE | X_MOVE, 10, "Dimension" }
                ATTR xxxx { 0, 0, 1, 0, 0, 0, 0.0 }
                FORM xxxx DR2D {
                        GRUP xxxx { 2 }
                        STXT xxxx { 0, 0.5, 1.0, 6.0, 5.0, 0.0, 4, "3.0" }
                        OPLY xxxx { 2, { 5.5, 5.5, 8.5, 5.5 } }
                }
        }
```

ProVector would call the ARexx script named Dimension if the user resized or moved this object. What exactly ProVector sends depends upon what the user does to the object. The following list shows what string(s) ProVector sends according to which flag(s) are set. The parameters are described below.

```
        X_CLONE      ``appl CLONE objID dx dy''
        X_MOVE       ``appl MOVE objID dx dy''
        X_ROTATE     ``appl ROTATE objID cx cy angle''
        X_RESIZE     ``appl RESIZE objID cx cy sx sy''
        X_CHANGE     ``appl CHANGE objID et ev ft fv ew jt fn''
        X_DELETE     ``appl DELETE objID''
        X_CUT        ``appl CUT objID''
        X_COPY       ``appl COPY objID''
        X_UNGROUP    ``appl UNGROUP objID''
```

where:

```
        appl is the name of the ARexx script
        CLONE, MOVE, ROTATE, RESIZE, etc. are literal strings
        objID is the object ID that ProVector assigns to this object
        (dx, dy) is the position offset of the CLONE or MOVE
        (cx, cy) is the point around which the object is rotated or resized
        angle is the angle (in degrees) the object is rotated
        sx and sy are the scaling factors in the horizontal and
          vertical directions, respectively.
        et is the edge type (the dash pattern index)
        ev is the edge value (the edge color index)
        ft is the fill type
        fv is the fill index
        ew is the edge weight
        jt is the join type
        fn is the font name
```

The X_CHANGE message reflects changes to the attributes found in the ATTR chunk.

If the user resized the XTRN object shown above by factor of 2, ProVector would call the ARexx script Dimension like this:

```
        Dimension RESIZE 1985427 7.0 4.75 2.0 2.0
```

The Object Chunks

The following chunks define the objects available in the DR2D FORM.

```
VBM  (0x56424D20)       /* Virtual BitMap */
```

The VBM chunk contains the position, dimensions, and file name of an ILBM image.

```
struct VBMstruct {
    IEEE        XPos, YPos,       /* Virtual coords */
                XSize, YSize,     /* Virtual size */
                Rotation;         /* in degrees */
    USHORT      PathLen;          /* Length of dir path */
    char        Path[PathLen];    /* Null-terminated path of file */
};
```

The coordinate (XPos, YPos) is the position of the upper left hand corner of the bitmap and the XSize and YSize fields supply the x and y dimensions to which the image should be scaled. Rotation tells how many degrees to rotate the ILBM around its upper left hand corner. ProVector does not currently support rotation of bitmaps and will ignore this value. Path contains the name of the ILBM file and may also contain a partial or full path to the file. DR2D readers should not assume the path is correct. The full path to an ILBM on one system may not match the path to the same ILBM on another system. If a DR2D reader cannot locate an ILBM file based on the full path name or the file name itself (looking in the current directory), it should ask the user where to find the image.

```
CPLY (0x43504C59)       /* Closed polygon */
OPLY (0x4F504C59)       /* Open polygon */
```

Polygons are the basic components of almost all 2D objects in the DR2D FORM. Lines, squares, circles, and arcs are all examples of DR2D polygons. There are two types of DR2D polygons, the open polygon (OPLY) and the closed polygon (CPLY). The difference between a closed and open polygon is that the computer adds a line segment connecting the endpoints of a closed polygon so that it is a continuous path. An open polygon's endpoints do not have to meet, like the endpoints of a line segment.

```
        struct POLYstruct {
            ULONG       ID;
            ULONG       Size;
            USHORT      NumPoints;
            IEEE        PolyPoints[2*NumPoints];
        };
```

The NumPoints field contains the number of points in the polygon and the PolyPoints array contains the (X, Y) coordinates of the points of the non-curved parts of polygons. The even index elements are X coordinates and the odd index elements are Y coordinates.

[Figure 3 - Bezier curves]

DR2D uses Bezier cubic sections, or cubic splines, to describe curves in polygons. A set of four coordinates (P1 through P4) defines the shape of a cubic spline. The first coordinate (P1) is the point where the curve begins. The line from the first to the second coordinate (P1 to P2) is tangent to the curve at the first point. The line from P3 to P4 is tangent to the cubic section, where it ends at P4.

The coordinates describing the cubic section are stored in the PolyPoints[] array with the coordinates of the normal points. DR2D inserts an indicator point before a set of cubic section points to differentiate a normal point from the points that describe a curve. An indicator point has an X value of 0xFFFFFFFF. The indicator point's Y value is a bit field. If this bit field's low-order bit is set, the points that follow the indicator point make up a cubic section.

The second lowest order bit in the indicator point's bit field is the MOVETO flag. If this bit is set, the point (or set of cubic section points) starts a new polygon, or subpolygon. This subpolygon will appear to be completely separate from other polygons but there is an important connection between a polygon and its subpolygon. Subpolygons make it possible to create holes in polygons. An example of a polygon with a hole is the letter ``O''. The ``O'' is a filled circular polygon with a smaller circular polygon within it. The reason the inner polygon isn't covered up when the outer polygon is filled is that DR2D fills are done using the even-odd rule.

The even-odd rule determines if a point is ``inside'' a polygon by drawing a ray outward from that point and counting the number of path segments the ray crosses. If the number is even, the point is outside the object and shouldn't be filled. Conversely, an odd number of crossings means the point is inside and should be filled. DR2D only applies the even-odd rule to a polygon and its subpolygons, so no other objects are considered in the calculations.

Taliesin, Inc. supplied the following algorithm to illustrate the format of DR2D polygons. OPLYs, CPLYs, AROWs, and ProVector's outline fonts all use the same format:

```
        typedef union {
            IEEE num;
            LONG bits;
        } Coord;

        #define INDICATOR       0xFFFFFFFF
        #define IND_SPLINE      0x00000001
        #define IND_MOVETO      0x00000002

        /* A common pitfall in attempts to support DR2D has
                been to fail to recognize the case when an
                INDICATOR point indicates the following
                coordinate to be the first point of BOTH a
                Bezier cubic and a sub-polygon, ie. the
                value of the flag = (IND_CURVE | IND_MOVETO) */
```

```
Coord   Temp0, Temp1;
int     FirstPoint, i, Increment;

/* Initialize the path */
NewPath();
FirstPoint = 1;

/* Draw the path */
i = 0;
while( i < NumPoints ) {
    Temp0.num = PolyPoints[2*i];    Temp1.num = PolyPoints[2*i + 1];
    if( Temp0.bits == INDICATOR ) {
        /* Increment past the indicator */
        Increment = 1;
        if( Temp1.bits & IND_MOVETO ) {
            /* Close and fill, if appropriate */
            if( ID == CPLY ) {
                FillPath();
            }
            else {
                StrokePath();
            }

            /* Set up the new path */
            NewPath();
            FirstPoint = 1;
        }
        if( Temp1.bits & IND_CURVE ) {
            /* The next 4 points are Bezier cubic control points */
            if( FirstPoint )
                MoveTo( PolyPoints[2*i + 2], PolyPoints[2*i + 3] );
            else
                LineTo( PolyPoints[2*i + 2], PolyPoints[2*i + 3] );
            CurveTo(    PolyPoints[2*i + 4], PolyPoints[2*i + 5],
                        PolyPoints[2*i + 6], PolyPoints[2*i + 7],
                        PolyPoints[2*i + 8], PolyPoints[2*i + 9] );
            FirstPoint = 0;
            /* Increment past the control points */
            Increment += 4;
        }
    }
    else {
        if( FirstPoint )
            MoveTo(     PolyPoints[2*i], PolyPoints[2*i + 1] );
        else
            LineTo(     PolyPoints[2*i], PolyPoints[2*i + 1] );
        FirstPoint = 0;

        /* Increment past the last endpoint */
        Increment = 1;
    }

    /* Add the increment */
    i += Increment;
}

/* Close the last path */
if( ID == CPLY ) {
    FillPath();
}
else {
    StrokePath();
}
```

```
GRUP (0x47525550)       /* Group */
```

The GRUP chunk combines several DR2D objects into one. This chunk is only valid inside nested DR2D FORMs, and must be the first chunk in the FORM.

```
        struct GROUPstruct {
            ULONG       ID;
            ULONG       Size;
            USHORT      NumObjs;
        };
```

The NumObjs field contains the number of objects contained in this group. Note that the layer of the GRUP FORM overrides the layer of objects within the GRUP. The following example illustrates the layout of the GRUP (and FILL) chunk.

```
       FORM { DR2D              /* Top-level drawing... */
                DRHD { ... }    /* Confirmed by presence of DRHD chunk */
                CMAP { ... }    /* Various other things... */
                FONS { ... }
                FORM { DR2D             /* A nested form... */
                        FILL { 1 }      /* Ah!  The fill-pattern table */
                        CPLY { ... }    /* with only 1 object */
                }
                FORM { DR2D             /* Yet another nested form */
                        GRUP { ..., 3 } /* Ah! A group of 3 objects */
                        TEXT { ... }
                        CPLY { ... }
                        OPLY { ... }
                }
                FORM { DR2D             /* Still another nested form */
                        GRUP { ..., 2 } /* A GRUP with 2 objects */
                        OPLY { ... }
                        TEXT { ... }
                }
        }
```

```
STXT (0x53545854)               /* Simple text */
```

The STXT chunk contains a text string along with some information on how and where to render the text.

```
        struct STXTstruct {
            ULONG       ID;
            ULONG       Size;
            UBYTE       Pad0;           /* Always 0 (for future expansion) */
            UBYTE       WhichFont;      /* Which font to use */
            IEEE        CharW, CharH,   /* W/H of an individual char */
                        BaseX, BaseY,   /* Start of baseline */
                        Rotation;       /* Angle of text (in degrees) */
            USHORT      NumChars;
            char        TextChars[NumChars];
        };
```

The text string is in the character array, TextChars[]. The ID of the font used to render the text is WhichFont. The font's ID is set in a FONS chunk. The starting point of the baseline of the text is (BaseX, BaseY). This is the point around which the text is rotated. If the Rotation field is zero (degrees), the text's baseline will originate at (BaseX, BaseY) and move to the right. CharW and CharH are used to scale the text after rotation. CharW is the average character width and CharH is the average character height. The CharW/H fields are comparable to an X and Y font size.

```
TPTH (0x54505448)               /* A text string along a path */
```

This chunk defines a path (polygon) and supplies a string to render along the edge of the path.

```
        struct TPTHstruct {
            ULONG   ID;
            ULONG   Size;
            UBYTE   Justification;      /* see defines, below */
            UBYTE   WhichFont;          /* Which font to use */
            IEEE    CharW, CharH;       /* W/H of an individual char    */
            USHORT  NumChars;           /* Number of chars in the string */
            USHORT  NumPoints;          /* Number of points in the path */
            char    TextChars[NumChars];/* PAD TO EVEN #! */
            IEEE    Path[2*NumPoints];  /* The path on which the text lies */
        };
```

WhichFont contains the ID of the font used to render the text. Justification controls how the text is justified on the line. Justification can be one of the following values:

```
        #define J_LEFT          0x00    /* Left justified */
        #define J_RIGHT         0x01    /* Right justified */
        #define J_CENTER        0x02    /* Center text */
        #define J_SPREAD        0x03    /* Spread text across path */
```

CharW and CharH are the average width and height of the font characters and are akin to X and Y font sizes, respectively. A negative FontH implies that the font is upsidedown. Note that CharW must not be negative. NumChars is the number of characters in the TextChars[] string, the string containing the text to be rendered. NumPoints is the number of points in the Path[] array. Path[] is the path along which the text is rendered. The path itself is not rendered. The points of Path[] are in the same format as the points of a DR2D polygon.

A Simple DR2D Example

Here is a (symbolic) DR2D FORM:

```
    FORM { DR2D
            DRHD 16 { 0.0, 0.0, 10.0, 8.0 }
            CMAP  6 { 0,0,0, 255,255,255 }
            FONS  9 { 1, 0, 1, 0, "Roman" } 0
            DASH 12 { 1, 2, {1.0, 1.0} }
            ATTR 14 { 0, 0, 1, 0, 0, 0, 0, 0.0 }
            BBOX 16 { 2.0, 2.0, 8.0, 6.0 }
            FORM { DR2D
                  GRUP  2 { 2 }
                  BBOX 16 { 3.0, 4.0, 7.0, 5.0 }
                  STXT 36 { 0,1, 0.5, 1.0, 3.0, 5.0, 0.0, 12, "Hello, World" }
                  BBOX 16 { 2.0, 2.0, 8.0, 6.0 }
                  OPLY 42 { 5, {2.0,2.0, 8.0,2.0, 8.0,6.0, 2.0,6.0, 2.0,2.0 }
            }
    }
```

[Figure 4 - Simple DR2D drawing]

```
The OFNT FORM

OFNT    (0x4F464E54)    /* ID of outline font file */

ProVector's outline fonts are stored in an IFF FORM called OFNT.  This
IFF is a separate file from a DR2D.  DR2D's FONS chunk refers only to
fonts defined in the OFNT form.


OFHD    (0x4F464844)    /* ID of OutlineFontHeaDer */

This chunk contains some basic information on the font.

        struct OFHDstruct {
            char    FontName[32];   /* Font name, null padded */
            short   FontAttrs;      /* See FA_*, below */
            IEEE    FontTop,        /* Typical height above baseline */
                    FontBot,        /* Typical descent below baseline */
                    FontWidth;      /* Typical width, i.e. of the letter O */
        };

        #define FA_BOLD         0x0001
        #define FA_OBLIQUE      0x0002
        #define FA_SERIF        0x0004

The FontName field is a NULL terminated string containing the name of
this font.  FontAttrs is a bit field with flags for several font attributes.
The flags, as defined above, are bold, oblique, and serif.  The unused
higher order bits are reserved for later use.  The other fields describe the
average dimensions of the characters in this font.  FontTop is the average
height above the baseline, FontBot is the average descent below the baseline,
and FontWidth is the average character width.


KERN    (0x4B45524C)    /* Kerning pair */

The KERN chunk describes a kerning pair.  A kerning pair sets the
distance between a specific pair of characters.

struct KERNstruct {
    short   Ch1, Ch2;           /* The pair to kern (allows for 16 bits...) */
    IEEE    XDisplace,          /* Amount to displace -left +right */
            YDisplace;          /* Amount to displace -down +up */
};

The Ch1 and Ch2 fields contain the pair of characters to kern.  These
characters are typically stored as ASCII codes.  Notice that OFNT stores
the characters as a 16-bit value.  Normally, characters are stored as 8-bit
values.  The wary programmer will be sure to cast assigns properly to avoid
problems with assigning an 8-bit value to a 16-bit variable.  The remaining
fields, XDisplace and YDisplace, supply the baseline shift from Ch1 to Ch2.


CHDF    (0x43484446)    /* Character definition */

This chunk defines the shape of ProVector's outline fonts.

struct CHDFstruct {
    short   Ch;           /* The character we're defining (ASCII) */
    short   NumPoints;    /* The number of points in the definition */
    IEEE    XWidth,       /* Position for next char on baseline - X */
            YWidth;       /* Position for next char on baseline - Y */
 /* IEEE    Points[2*NumPoints]*/       /* The actual points */
};
```

```
#define INDICATOR       0xFFFFFFFF      /* If X == INDICATOR, Y is an action */
#define IND_SPLINE      0x00000001      /* Next 4 pts are spline control pts */
#define IND_MOVETO      0x00000002         /* Start new subpoly */
#define IND_STROKE      0x00000004      /* Stroke previous path */
#define IND_FILL        0x00000008      /* Fill previous path */

Ch is the value (normally ASCII) of the character outline this chunk
defines.  Like Ch1 and Ch2 in the KERN chunk, Ch is stored as a 16-bit
value.  (XWidth,YWidth) is the offset to the baseline for the
following character.  OFNT outlines are defined using the same method
used to define DR2D's polygons (see the description of OPLY/CPLY for
details).

Because the OFNT FORM does not have an ATTR chunk, it needed an
alternative to make fills and strokes possible.  There are two extra
bits used in font indicator points not found in polygon indicator
points, the IND_STROKE and IND_FILL bits (see defines above).  These
two defines describe how to render the current path when rendering
fonts.

The current path remains invisible until the path is either filled
and/or stroked.  When the IND_FILL bit is set, the currently defined
path is filled in with the current fill pattern (as specified in the
current ATTR chunk).  A set IND_STROKE bit indicates that the
currently defined path itself should be rendered.  The current ATTR's
chunk dictates the width of the line, as well as several other
attributes of the line.  These two bits apply only to the OFNT FORM
and should not be used in describing DR2D polygons.
```

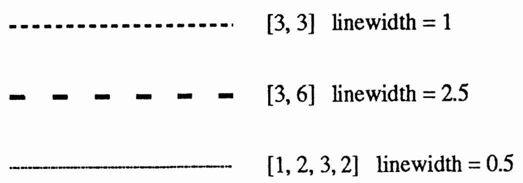


Figure 1 - Dash Patterns

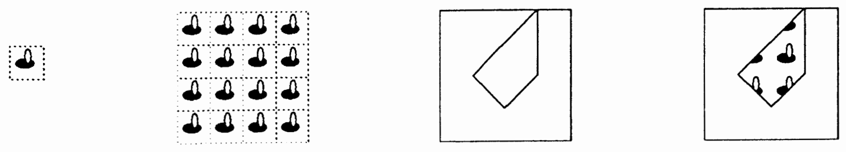


Fig. A Fig. B Fig. C Fig. D

Figure 2 - Fill Patterns

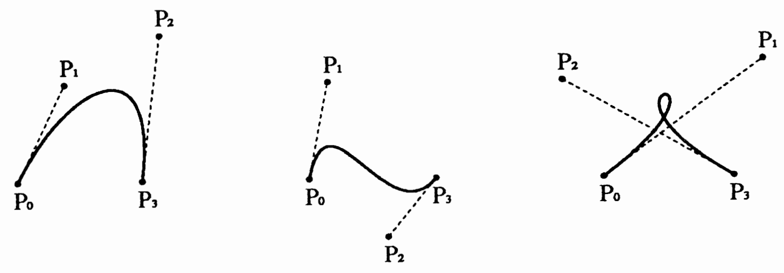


Figure 3 - Beziér Curves

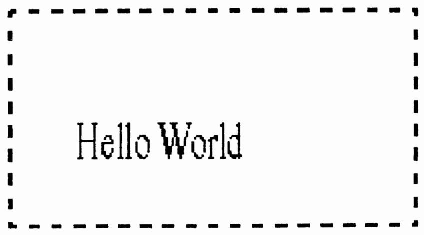


Figure 4 - Simple DR2D Drawing

FANT.doc Page 1

```
Fantavision movie format

FORM FANT

/************************************************************************/
/*                                                                    **
** - FantForm.h                                                       **
**                                                                    **
**     This is the IFF movie format for Amiga Fantavision.            **
**                                                                    **
**     (c) Copyright 1988 Broderbund Software                         **
**                                                                    **
**     - FORMAT FROZED May 5, 1988 -                                  **
**                                                                    **
**     Implemented by Steve Hales                                     **
**                                                                    **
** Overvue -                                                          **
**     This is a description of the format used for Amiga             **
**     Fantavision.  It assumes you have intimate knowledge of how    **
**     IFF-FORMs are constructed, layed out, and read.  This file     **
**     can be used as a header file.  This is fairly complete, but    **
**     I'm sure there are a few things missing.                       **
**                                                                    **
**     I can be reached in the following ways:                        **
**       UseNet:    Steve_A_Hales@cup.portal.com  OR                  **
**                  sun!cup.portal.com!Steve_A_Hales                  **
**                                                                    **
**       US Mail:   882 Hagemann Drive                                **
**                  Livermore, CA, 94550-2420                         **
**                                                                    **
**       Phone:     (415) 449-5297                                    **
**                                                                    **
**     NOTE:  I cannot, by contract, give out any code to load or     **
**            play Fantavision movies.  If that is want you want      **
**            then you will need to contact Broderbund Software       **
**            directly.  Their number is (415) 492-3200.              **
**                                                                    **
** Enjoy!  Aloha.                                                     **
**                                                                    */
/************************************************************************/

/* Misc Fantavision structures
*/
typedef struct Rect
{
   int left, top, right, bottom;
};
typedef struct Point
{
   int h, v;
};

/* Frame opcodes */
#define opNEXT     0       /* go on to next frame */
#define opREPEAT   1       /* repeat sequence starting from frame Parm repl times *
/
#define opGOTO     2       /* goto frame Parm */

/* Frame modes */
#define fNORMAL    0x0000  /* redraw every frame */
#define fTRACE     0x0001  /* draw into both paged screens */
#define fLIGHTNING 0x0020  /* don't erase background */
```

FANT.doc Page 2

```
/* Fantavision FORM defines
*/
#define ID_FANT    'FANT'          /* FORM type */
#define ID_FHDR    'FHDR'          /* Movie Header */
#define ID_FRAM    'FRAM'          /* Format info for a Frame */
#define ID_POLY    'POLY'          /* Format info for a Polygon */
#define ID_CSTR    'CSTR'          /* \0 terminated string */

/* Polygon modes */
#define pTYPEMASK  0x00FF      /* type mask to get just type of poly */
#define pSELECT    0x8000      /* is object selected? */
#define pOUTLINE   0x4000      /* outlined polygon using DotModeSide to
                               ** determine when to not connect a line.
                               ** ex. 0 draws on all sides, 1 will draw on
                               ** everyother side, 2 will leave every second
                               ** side blank, 3 will every third side
                               ** blank, etc. */
#define pBACKDROP  0x2000      /* polygon will be dropped into the background
                               ** during animation. */
#define pMSKBITMAP 0x1000      /* bitmap has a mask */

/* Polygon types */
#define pDELETE    0x7000      /* object is a filler (its deleted from display) */
#define pFILLED    0           /* filled polygon */
#define pLINE      1           /* not-connected line polygon */
#define pLINED     2           /* connected line polygon */
#define pTEXTBLOCK 3           /* text block to draw */
#define pCIRCLEDOT 4           /* draw circle dots at vertex's using
                               ** dotSize at size. */
#define pRECTDOT   5           /* draw square dots at vertex's using
                               ** dotSize at size. */
#define pBITMAPDOT 6           /* draw dots using a bitmap at vertex's using
                               ** BitMap. */
#define pBITMAP    7           /* draw just bitmap image */   '

/* These are used for the pTEXTBLOCK polygon type
*/
/* Text justification
*/
#define tLEFT      0
#define tCENTER    1
#define tRIGHT     2
/* Text style
*/
#define tNORMAL    (int)(FS_NORMAL)
#define tBOLD      (int)(FSF_BOLD)
#define tITALIC    (int)(FSF_ITALIC)
#define tUNDERLINE (int)(FSF_UNDERLINED)
#define tEXTENDED  (int)(FSF_EXTENDED)


/* Fantavision movie header -
**
**     This header defines how much RAM is needed, how many frames, and sounds
**     in the movie.
*/
typedef struct FantHeader
{
   int PointsPerObj;       /* number of vertexs per object */
   int ObjsPerFrame;       /* number of objects per frame */
   int ScreenDepth;        /* 0 to 6, for number of bit planes */
   int ScreenWidth;        /* in pixels */
   int ScreenHeight;       /* in pixels */
   int BackColor;          /* background color palette number */
   long SizeOfMovie;       /* RAM Size of movie, expanded */
```

```
    int pad[30];            /* padding for expanding */
    int NumberOfFrames;
    int NumberOfSounds;
    int NumberOfBitMaps;
    int Background;         /* non-zero if first bitmap is a background */
    int SpeedOfMovie;       /* 100 is normal speed, 50 is half speed, etc */
    int pad[3];             /* expansion */
};

/* Fantavision frame info -
**
**      Each frame has this structure defined.
*/
typedef struct FrameFormat
{
    int OpCode;                 /* Frame opcode */
    long Parm;                  /* contains frame number for opNEXT, opREPEAT */
    char Rep1, Rep2;            /* Rep1 is repeat counter, Rep2 is not used */
    int TweenRate;              /* number of tweens per frame */

    int ChannelIndex[2];        /* -3 stop sound is this channel
                                ** -2 modify current sound
                                ** -1 no sound for this channel
                                ** (all others) is an index into the sound
                                ** list.  Which sound to use.
                                */

    int NumberOfPolys;          /* number of polygons in this frame */
    int ColorPalette[32];       /* xRGB - format 4 bits per register */
    int Pan, Tilt;              /* 0 is centered, (+-) amounts are in pixels */
    int Modes;                  /* Frame modes */
    int pad;                    /* expansion */
};

/* Fantavision polygon info -
**
**      Each polygon has this structure defined.
*/
typedef struct PolyFormat
{
    int NumberOfPoints;         /* how many vertexs for this polygon */
    int Type;                   /* polygon type */
    int Color;                  /* palette color number (see note 1) */
    Rect Bounds;                /* enclosing rectangle of polygon */
    int Depth;                  /* polygon view depth (see note 2) */
    char DotModeSize;           /* in pixels, not larger than 40 */
    char DotModeSide;           /* determines outlining features */
    int OutlineColor;           /* palette color number for outline */
    int BitMapIndex;            /* if not -1, then bitmap index into bitmap list */
    int BMRealWidth;            /* in pixels */
    int BMRealHeight;
    int TextLength;             /* length of text for pTEXTBLOCK */
    int TextJust;
    char TextSize;              /* size in pixels */
    char TextStyle;
    long pad;                   /* expansion */
    Point p[];                  /* array of points defining vertexs */
};
```

```
/* Fantavision high-level IFF format.
**
   FORM FANT
       FHDR

       - background -
       FORM ILBM   if Background is non-zero
           BMHD
           BODY

       - bitmap list -
       NOTE:  If a bitmap has a mask, it will be compute during load time.

       FORM ILBM   times NumberOfBitMaps
           BMHD
           BODY

       - sound list -
   {   FORM 8SVX   times NumberOfSounds
           VHDR
           BODY
       SEFX    }   Default parameters for sound

       - frame list -
   {   FRAM        times NumberOfFrames
       SEFX        if sound for channel 1.
       SEFX        if sound for channel 2.
       POLY        times NumberOfPolys
     { CSTR        Text of poly if PolyType = pTEXTBLOCK
       CSTR   } }  Name of font
*/

/******- NOTES -***************************************************************/
/*
** 1 - The color palette is a number from 0 to 1120.  The first 32 numbers
**     are normal RGB colors, but the rest index into a pre-defined
**     set of patterns.
**
** 2 - The view depth of each polygon determines the display order.  The
**     higher the number the closer the polygon is to the viewer.  During
**     editing, each polygon is assigned numbers in multiples of 100,
**     but to function, any number can work.
*/
```

HEAD.doc

Flow - New Horizons Software

TITLE: HEAD (FORM used by Flow - New Horizons Software, Inc.)

IFF FORM / CHUNK DESCRIPTION
============================

Form/Chunk ID: FORM HEAD, Chunks NEST, TEXT, FSCC

Date Submitted: 03/87
Submitted by: James Bayless - New Horizons Software, Inc.

FORM
====

FORM ID: HEAD

FORM Description:

FORM HEAD is the file storage format of the Flow idea processor by New Horizons Software, Inc. Currently only the TEXT and NEST chunks are used. There are plans to incorporate FSCC and some additional chunks for headers and footers.

CHUNKS
======

CHUNK ID: NEST

This chunk consists of only of a word (two byte) value that gives the new current nesting level of the outline. The initial nesting level (outermost level) is zero. It is necessary to include a NEST chunk only when the nesting level changes. Valid changes to the nesting level are either to decrease the current value by any amount (with a minimum of 0) or to increase it by one (and not more than one).

CHUNK ID: TEXT

This chunk is the actual text of a heading. Each heading has a TEXT chunk (even if empty). The text is not NULL terminated - the chunk size gives the length of the heading text.

CHUNK ID: FSCC

This chunk gives the Font/Style/Color changes in the heading from the most recent TEXT chunk. It should occur immediately after the TEXT chunk it modifies. The format is identical to the FSCC chunk for the IFF form type 'WORD' (for compatibility), except that only the 'Location' and 'Style' values are used (i.e., there can be currently only be style changes in an outline heading). The structure definition is:

```
typedef struct {
   UWORD   Location;   /* Char location of change */
   UBYTE   FontNum;    /* Ignored */
   UBYTE   Style;      /* Amiga style bits */
   UBYTE   MiscStyle;  /* Ignored */
   UBYTE   Color;      /* Ignored */
   UWORD   pad;        /* Ignored */
} FSCChange;
```

The actual chunk consists of an array of these structures, one entry for each Style change in the heading text.

ILBM.CLUT.doc

Color Lookup Table chunk

amiga.dev/iff message 1527

TITLE: CLUT IFF chunk proposal

"CLUT" IFF 8-Bit Color Look Up Table

Date: July 2, 1989
From: Justin V. McCormick
Status: Public Proposal
Supporting Software: FG 2.0 by Justin V. McCormick for PP&S

Introduction:

This memo describes the IFF supplement for the new chunk "CLUT".

Description:

A CLUT (Color Look Up Table) is a special purpose data module containing table with 256 8-bit entries. Entries in this table can be used directly as a translation for one 8-bit value to another.

Purpose:

To store 8-bit data look up tables in a simple format for later retrieval. These tables are used to translate or bias 8-bit intensity, contrast, saturation, hue, color registers, or other similar data in a reproducable manner.

Specifications:

```
/* Here is the IFF chunk ID macro for a CLUT chunk */
#define ID_CLUT MakeID('C','L','U','T')

/*
 * Defines for different flavors of 8-bit CLUTs.
 */
#define CLUT_MONO       0L      /* A Monochrome, contrast or intensity LUT */
#define CLUT_RED        1L      /* A LUT for reds                          */
#define CLUT_GREEN      2L      /* A LUT for greens                        */
#define CLUT_BLUE       3L      /* A LUT for blues                         */
#define CLUT_HUE        4L      /* A LUT for hues                          */
#define CLUT_SAT        5L      /* A LUT for saturations                   */
#define CLUT_UNUSED6    6L      /* How about a Signed Data flag            */
#define CLUT_UNUSED7    7L      /* Or an Assumed Negative flag             */

/* All types > 7 are reserved until formally claimed */
#define CLUT_RESERVED_BITS 0xfffffff8L

/* The struct for Color Look-Up-Tables of all types */
typedef struct
{
  ULONG type;             /* See above type defines */
  ULONG res0;             /* RESERVED FOR FUTURE EXPANSION */
  UBYTE lut[256];         /* The 256 byte look up table */
} ColorLUT;
```

CLUT Example:

Normally, the CLUT chunk will appear after the BMHD of an FORM ILBM before the BODY chunk, in the same "section" as CMAPs are normally found. However, a FORM may contain only CLUTs with no other supporting information.

As a general guideline, it is desirable to group all CLUTs together in a form without other chunk types between them. If you were using CLUTs to store RGB intensity corrections, you would write three CLUTs in a row, R, G, then B.

Here is a box diagram for a 320x200x8 image stored as an IFF ILBM with a single CLUT chunk for intensity mapping:

```
+-----------------------------------+
|'FORM'         64284               |     FORM 64284 ILBM
+-----------------------------------+
|'ILBM'                             |
+-----------------------------------+
| +-------------------------------+ |
| | 'BMHD'      20                | |     .BMHD 20
| | 320, 200, 0, 0, 8, 0, 0, ...  | |
| | ------------------------------+ |
| | 'CLUT'      264               | |     .CLUT 264
| | 0, 0, 0; 32, 0, 0; 64,0,0; .. | |
| +-------------------------------+ |
| +-------------------------------+ |
| |'BODY'               64000     | |     .BODY 64000
| |0, 0, 0, ...                   | |
| +-------------------------------+ |
+-----------------------------------+
```

Design Notes:

I have deliberately kept this chunk simple (KISS) to facilitate implementation. In particular, no provision is made for expansion to 16-bit or 32-bit tables. My reasoning is that a 16-bit table can have 64K entries, and thus would benefit from data compression. My suggestion would be to propose another chunk or FORM type better suited for large tables rather than small ones like CLUT.

Newtek Dynamic Ham color chunks

Newtek for Digiview IV (dynamic Ham)

ILBM.DYCP - dynamic color palette
3 longwords (file setup stuff)

ILBM.CTBL - array of words, one for each color (0rgb)

```
Dots per inch chunk

ILBM DPI chunk
==============

Registered by:

Spencer Shanson
16 Genesta Rd
Plumstead
London SE18 3ES
ENGLAND

1-16-90

ILBM.DPI   Dots Per Inch   to allow output of an image at the
same resolution it was scanned at

typedef struct {
        UWORD dpi_x;
        UWORD dpi_y;
        } DPIHeader ;

For example, an image scanned at horizontal resolution of
240dpi and vertical resolution of 300dpi would be saved as:

44504920 00000004 00F0 012C
D P I    size     dpi_x dpi_y
```

```
DPaint perspective chunk (EA)

IFF FORM / CHUNK DESCRIPTION
============================

Form/Chunk ID:   Chunk DPPV  (DPaint II ILBM perspective chunk)
Date Submitted:  12/86
Submitted by:    Dan Silva

Chunk Description:

   The DPPV chunk describes the perspective state in a DPaintII ILBM.

Chunk Spec:

/* The chunk identifier DPPV */
#define ID_DPPV    MakeID('D','P','P','V')

typedef LONG LongFrac;
typedef struct ( LongFrac x,y,z; )  LFPoint;
typedef LongFrac  APoint[3];

typedef union {
   LFPoint l;
   APoint  a;
   } UPoint;

/* values taken by variable rotType */
#define ROT_EULER  0
#define ROT_INCR   1

/* Disk record describing Perspective state */

typedef struct {
   WORD     rotType;            /* rotation type */
   WORD     iA, iB, iC;         /* rotation angles (in degrees) */
   LongFrac Depth;              /* perspective depth */
   WORD     uCenter, vCenter;   /* coords of center perspective,
                                 * relative to backing bitmap,
                                 * in Virtual coords
                                 */
   WORD     fixCoord;           /* which coordinate is fixed */
   WORD     angleStep;          /* large angle stepping amount */
   UPoint   grid;               /* gridding spacing in X,Y,Z */
   UPoint   gridReset;          /* where the grid goes on Reset */
   UPoint   gridBrCenter;       /* Brush center when grid was last on,
                                 * as reference point
                                 */
   UPoint   permBrCenter;       /* Brush center the last time the mouse
                                 * button was clicked, a rotation performed,
                                 * or motion along "fixed" axis
                                 */
   LongFrac rot[3][3];          /* rotation matrix */
   } PerspState;

SUPPORTING SOFTWARE
===================
DPaint II   by Dan Silva for Electronic Arts
DPaint IV enhanced color cycle chunk (EA)

DRNG Chunk for FORM ILBM
========================

Submitted by Lee Taran

Purpose:

Enhanced Color Cycling Capabilities
-------------------------------------
   * DPaintIV supports a new color cycling model which does NOT
     require that color cycles contain a contiguous range of color
     registers.

     For example:
       If your range looks like:  [1][3][8][2]
       then at each cycle tick
          temp = [2],
          [2] = [8],
          [8] = [3],
          [3] = [1],
          [1] = temp

   * You can now cycle a single register thru a series of rgb values.
     For example:
        If your range looks like: [1] [orange] [blue] [purple]
        then at each cycle tick color register 1 will take on the
        next color in the cycle.

        ie:  t=0:  [1] = curpal[1]
             t=1:  [1] = purple
             t=2:  [1] = blue
             t=3:  [1] = orange
             t=4:  goto t=0

   * You can combine rgb cycling with traditional color cycling.
     For example:
         Your range can look like:
             [1] [orange] [blue] [2] [green] [yellow]

         t=0: [1] = curpal[1], [2] = curpal[2]
         t=1: [1] = yellow,    [2] = blue
         t=2: [1] = green,     [2] = orange
         t=3: [1] = curpal[2], [2] = curpal[1]
         t=4: [1] = blue,      [2] = yellow
         t=5: [1] = orange,    [2] = green
         t=6: goto t=0

Note:
   * DPaint will save out an old style range CRNG if the range fits
     the CRNG model otherwise it will save out a DRNG chunk.
   * no thought has been given (yet) to interlocking cycles
```

```
/* ----------------------------------------------------------------------

 IFF Information:  DPaintIV DRNG chunk

         DRNG ::= "DRNG" # { DRange DColor* DIndex* }

 a <cell> is where the color or register appears within the range

 The RNG_ACTIVE flags is set when the range is cyclable. A range
 should only have the RNG_ACTIVE if it:
      1> contains at least one color register
      2> has a defined rate
      3> has more than one color and/or color register
 If the above conditions are met then RNG_ACTIVE is a user/program
 preference.  If the bit is NOT set the program should NOT cycle the
 range.

 The RNG_DP_RESERVED flags should always be 0!!!
 ---------------------------------------------------------------------- */
typedef struct {
   UBYTE min;           /* min cell value */
   UBYTE max;           /* max cell value */
   SHORT rate;          /* color cycling rate, 16384 = 60 steps/second */
   SHORT flags;         /* 1=RNG_ACTIVE,4=RNG_DP_RESERVED */
   UBYTE ntrue;         /* number of DColor structs to follow */
   UBYTE nregs;         /* number of DIndex structs to follow */
   } DRange;

typedef struct { UBYTE cell; UBYTE r,g,b; } DColor; /* true color cell */
typedef struct { UBYTE cell; UBYTE index; } DIndex; /* color register cell */
```

ILBM.EPSF.doc

```
Encapsulated Postscript chunk

ILBM EPSF Chunk
===============

Pixelations   Kevin Saltzman   617-277-5414

Chunk to hold encapsulated postscript

Used by PixelScript in their clip art.  Holds a postscript
representation of the ILBM's graphic image.

EPSF length
   ; Bounding box
   WORD lowerleftx;
   WORD lowerlefty;
   WORD upperrightx;
   WORD upperrighty;
   CHAR []    ; ascii postscript file
```

MTRX.doc

```
Numerical data storage (MathVision - Seven Seas)

MTRX FORM, for matrix data storage                    19-July-1990

Submitted by: Doug Houck
              Seven Seas Software
              (address, etc)

INTRODUCTION:

Numerical data, as it comes from the real world, is an ill-mannered beast.
Often much is assumed about the data, such as the number of dimensions,
formatting, compression, limits, and sizes.  As such, data is not portable.
The MTRX FORM will both store the data, and completely describe its
format, such that programs no longer need to guess the parameters of
a data file.  There needs to be but one program to read ascii files and
output MTRX IFF files.

A matrix, by our definition, is composed of three types of things.
Firstly, the atomic data, such as an integer, or floating point number.
Secondly, arrays, which are simply lists of things which are all the same.
Thirdly, structures, which are lists of things which are different.
Both arrays and structures may be composed of things besides atomic data -
they may contain other structures and arrays as well.  This concept
of nesting structures may be repeated to any desired depth.

For example, a list of data pairs could be encoded as an array of structures,
where each structure contains two numbers.  A two-dimensional array is
simply an array of arrays.

Since space conservation is often desirable, there is provision for
representing each number with fewer bits, and compressing the bits together.


CHUNKS

The MTRX FORM is composed of the definition of the structure, followed
by the BODY which contains the data which is defined.  Usually, there
is only one set of data, but a smarter IFF read could use the definition
as a PROPerty, with identically formatted data sets (BODYs) in a LIST.

  FORM MTRX
    definition (ARRY | STRU | DTYP)
    BODY

ARRY: The array chunk defines a counted list of similar items.
The first (required) chunk in an ARRY is ELEM, which gives the number
of elements in the array.  Optionally, there may be limits given, (LOWR
and UPPR), which could be used in scaling during sampling of the data.
Lastly is the definition of an element of the array, which may be a
nested definition like everything else.

  ARRY ::= "ARRY" #{ ELEM [LOWR] [UPPR] [PACK] ARRY|STRU|DTYP }

STRU: The structure chunk defines a counted list of dissimilar things.
The first (required) chunk in a STRU is FLDS, which gives the number
of fields in the structure.  Lastly are definitions of each field
in the structure.  Again, each field may have a nested definition like
everything else.

 STRU ::= "STRU" #{ FLDS ([PACK] ARRY|STRU|DTYP)* }

VALU: The value contains a datatype, and then a constant of that type.
The datatype contains the size of the constant, so this chunk has variable
size.  VALU is used in the ARRY chunk to give the scaling limits of the array.
```

```
BODY: This is the actual data we went to so much effort to describe.
It is stored in "row-first" format, that is, items at the bottom of the
nested description are stored next to each other.  In most cases, it
should be sufficient to simply block-read the whole chunk from disk,
unless the reader needs to adjust byte-ordering or store in a more
time-efficient format in memory.  Data is assumed to be byte-aligned.

PACK: The PACK chunk is necessary when the bit length of the data is
not a multiple of 8, that is, not byte-aligned, and the user wishes
to conserve space by packing data items together.  PACK is simply a
number - the number of items to bit-pack before aligning on a byte.
A PACK is in effect for the remainder of its nested scope, or until
overridden by a new specification.  A STRU or ARRY is assumed to have
a PACK of 1 by default - it is not affected by PACKs in definitions above.
A PACK of 0 means to byte-align before processing the next definition.
The PACK specifier should be normalized.  For example, when packing a large
array of 3-bit numbers, PACK should be 8 since 3*8 = 24. In this case 8 is
the smallest PACK number which aligns on a byte naturally.

DTYP: The DataType is the most interesting chunk, as it attempts to define
every conceivable type of numeric data with 32 bits.  The 32 bits are broken
down into three fields, 1) the size in bits, 2) the Class, and 3) SubClass.
The Class makes the most major distinction, separating integers from floating
point numbers from Binary Coded Decimal and etc.  Within each class is a
SubClass, which gives the specific encoding used.  Finally, the Size tells
what how much room the data occupies.  The basic division of datatypes is
given in the tree structure below.

Class             SubClass     Size  Final Specific Type
=====             ========     ====  ==================
 |
Binary Unsigned - 0 ------------ 8   UByte
 |                              16   UWord
 |                              32   ULong
 |
Binary Signed --- 0 ------------  8  Byte
 |                               16  Word
 |                               32  ULong
 |
Real ------------Ieee38 -------- 32  Ieee Single Precision
 |                 |
 |               Ieee308 ------- 64  Double Precision
 |                 |             32  Truncated Double Precision
 |                 |
 |               FFP ----------- 32  Motorola Fast Floating Point
 |
Text ----------- Text0 --------- ??  Null-terminated text
 |                 |
 |               CText --------- ??  Number of characters in first byte
 |                 |
 |               FText --------- ??  Fixed length, space padded
 |
BCD ------------ Nibble -------- ??
                   |
                 Character ----- ??

A design goal was to create a classification system which other people
can easily plug into.  Many data types are simply size variations on
existing data types.  For example, a 4-bit integer can be specified by
giving the size as four bits in the Signed Binary class.  Be aware that
not all MTRX readers may support your new type, but there will not be
any type clashes or ambiguities by following these rules.  If you have
a truly unique Class or SubClass, you will need to register it with
Commodore to prevent clashes.
```

```
A second design goal was to create a format which is easily decoded
by software.  By aligning fields on bytes, you have the option of redefining
the datatype as a structure, so as to avoid shifting when accessing the
fields.  Since the numbers are sequentially assigned, they are suitable
as array indicies, and may be optimized in a C switch statement.

A third design goal was allowing for naive and sophisticated readers.
In checking for a certain datatype, a naive reader can simply compare
the whole datatype with a small set of known types, which assumes that
each different Size defines a unique datatype.  Sophisticated readers
will consider the Class, SubClass and Size separately, so as to support
arbitrary size integers, and truncated Floating Point numbers, for example.

 *
 * MTRX ::= "FORM" #{ "MTRX" ARRY|STRU|DTYP BODY        } Matrix
 * ARRY ::= "ARRY" #{ ELEM [LOWR] [UPPR] [PACK] ARRY|STRU|DTYP } Array
 * STRU ::= "STRU" #{ FLDS ([PACK] ARRY|STRU|DTYP)*     } Structure
 * ELEM ::= "ELEM" #{ elements                          } Array elements
 * LOWR ::= "LOWR"  { VALU                              } Minimum limit
 * UPPR ::= "UPPR"  { VALU                              } Maximum limit
 * VALU ::=        #{ dtyp value                        } Value (in union)
 * dtyp ::=         { size, subclass, class             } Data Type (scalar)
 * DTYP ::= "DTYP" #{ dtyp                              }
 * FLDS ::= "FLDS" #{ number of fields                  } Number of Fields
 * PACK ::= "PACK" #{ units packed b4 byte alignment    } Packing
 * BODY ::= "BODY" #{ inner-first binary dump           } Data
 *
 *   [] means optional
 *   #  means the size of the unit following
 *   *  means one or more of
 *
```

PGTB.doc Page 1

```
Program traceback (SAS Institute)

FORM PGTB

Proposal:
        New IFF chunk type, to be named PGTB, meaning ProGram TraceBack.

Format:

        'PGTB'          - chunk identifier
        length          - longword for length of chunk

        'FAIL'          - subfield giving environment at time of crash
        length          - longword length of subfield
        NameLen         - length of program name in longwords (BSTR)
        Name            - program name packed in longwords
        Environment     - copy of AttnFlags field from ExecBase,
                          gives type of processor, and existence of
                          math chip
        VBlankFreq      - copy of VBlankFrequency field from ExecBase
        PowerSupFreq    - copy of PowerSupplyFrequency field from ExecBase
                          above fields may be used to determine whether
                          machine was PAL or NTSC
        Starter         - non-zero = CLI, zero = WorkBench
        GURUNum         - exception number of crash
        SegCount        - number of segments for program
        SegList         - copy of seglist for program
                          (Includes all seglist pointers, paired with
                           sizes of the segments)

        'REGS'          - register dump subfield
        length          - length of subfield in longwords
        GURUAddr        - PC at time of crash
        Flags           - copy of Condition Code Register
        DDump           - dump of data registers
        ADump           - dump of address registers

        'VERS'          - revision of program which created this file
        length          - length of subfield in longwords
        version         - main version of writing program
        revision        - minor revision level of writing program
        TBNameLen       - length of name of writing program
        TBName          - name of writing program packed in longwords (BSTR)

        'STAK'          - stack dump subfield
        length          - length of subfield in longwords
        (type)          - tells type of stack subfield, which can be any of
                          the following:
                ----------------------------------------------------------
                Info            - value 0
                StackTop        - address of top of stack
                StackPtr        - stack pointer at time of crash
                StackLen        - number of longwords on stack

                ----------------------------------------------------------
                Whole stack     - value 1
                                  only used if total stack to be dumped is 8k
                                  or less in size
                Stack           - dump of stack from current to top

                ----------------------------------------------------------
                Top 4k          - value 2
                                  if stack used larger than 8k, this part
                                  is a dump of the top 4k
                Stack           - dump of stack from top - 4k to top
```

PGTB.doc Page 2

```
                ----------------------------------------------------------
                Bottom 4k       - value 3
                                  if stack used larger than 8k, this part
                                  is a dump of the bottom 4k
                Stack           - dump of stack from current to current + 4k

        In other words, we will dump a maximum of 8k of stack data.  This
        does NOT mean the stack must be less than 8k in size to dump the
        entire stack, just that the amount of stack USED be less than 8k.

        'UDAT'          - Optional User DATa chunk.  If the user assigns
                          a function pointer to the label "_ONGURU", the
                          catcher will call this routine prior to closing
                          the SnapShot file, passing one parameter on the
                          stack - an AmigaDOS file pointer to the SnapShot
                          file.  Spec for the _ONGURU routine:

                                void <function name>(fp)
                                long fp;

                          In other words, your routine must be of type 'void'
                          and must take one parameter, an AmigaDOS file
                          handle (which AmigaDOS wants to see as a LONG).
        length          - length of the UserDATa chunk, calculated after the
                          user routine terminates.
```

```
DPaint IV perspective move form (EA)

Submitted by Lee Taran

/* ----------------------------------------------------------------------
   IFF Information:
       PRSP ::= "FORM" # {"PSRP" MOVE }
       MOVE ::= "MOVE" # { MoveState  }
 * ---------------------------------------------------------------------- */
typedef struct {
   BYTE reserved;          /* initialize to 0 */
   BYTE moveDir;           /* 0 = from point  1 = to point  */
   BYTE recordDir;         /* 0 = FORWARD,    1 = STILL, 2 = BACKWARD */
   BYTE rotationType;      /* 0 = SCREEN_RELATIVE, 1 = BRUSH_RELATIVE */
   BYTE translationType;   /* 0 = SCREEN_RELATIVE, 1 = BRUSH_RELATIVE */
   BYTE cyclic;            /* 0 = NO, 1 = YES */
   SHORT distance[3];      /* x,y,z distance displacement */
   SHORT angle[3];         /* x,y,z rotation angles */
   SHORT nframes;          /* number of frames to move */
   SHORT easeout;          /* number of frames to ease out */
   SHORT easein;           /* number of frames to ease in */
   } MoveState;
```

```
RGB image forms, Turbo Silver (Impulse)

                    FORM RGBN and FORM RGB8
                    -----------------------

        RGBN and RGB8 files are used in Impulse's Turbo Silver and Imagine.
        They are almost identical to FORM ILBM's except for the BODY chunk
        and slight differences in the BMHD chunk.

        A CAMG chunk IS REQUIRED.

        The BMHD chunk specfies the number of bitplanes as 13 for type RGBN
        and 25 for type RGB8, and the compression type as 4.

        The FORM RGBN uses 12 bit RGB values, and the FORM RGB8 uses
        24 bit RGB values.

        The BODY chunk contains RGB values, a "genlock" bit, and repeat
        counts.  In Silver, when "genlock" bit is set, a "zero color" is
        written into the bitplanes for genlock video to show through.
        In Diamond and Light24 (Impulse 12 & 24 bit paint programs),
        the genlock bit is ignored if the file is loaded as a picture
        (and the RGB color is used instead), and if the file is loaded
        as a brush the genlock bit marks pixels that are not part of
        the brush.

        For both RGBN and RGB8 body chunks, each RGB value always has a
        repeat count.  The values are written in different formats depending
        on the magnitude of the repeat count.

        For the RGBN BODY chunk:

                For each RGB value, a WORD (16-bits) is written: with the
                12 RGB bits in the MSB (most significant bit) positions;
                the "genlock" bit next; and then a 3 bit repeat count.
                If the repeat count is greater than 7, the 3-bit count is
                zero, and a BYTE repeat count follows.  If the repeat count
                is greater than 255, the BYTE count is zero, and a WORD
                repeat count follows.  Repeat counts greater than 65536 are
                not supported.

        For the RGB8 body chunk:

                For each RGB value, a LONG-word (32 bits) is written:
                with the 24 RGB bits in the MSB positions; the "genlock"
                bit next, and then a 7 bit repeat count.

                In a previous version of this document, there appeared the
                following line:

                "If the repeat count is greater than 127, the same rules apply
                as in the RGBN BODY."

                But Impulse has never written more than a 7 bit repeat count,
                and when Imagine and Light24 were written, they didn't support
                reading anything but 7 bit counts.
```

Sample BODY code:

```
                if(!count) {
                        if (Rgb8) {
                                fread (&w, 4, 1, RGBFile);
                                lock = w & 0x00000080;
                                rgb = w >> 8;
                                count = w & 0x0000007f;
                        } else {
                                w = (UWORD) getw (RGBFile);
                                lock = w & 8;
                                rgb = w >> 4;
                                count = w & 7;
                        }
                        if (!count)
                                if (!(count = (UBYTE) getc (RGBFile)))
                                        count = (UWORD) getw (RGBFile);
                }
```

The pixels are scanned from left to right across horizontal lines, processing from top to bottom. The (12 or 24 bit) RGB values are stored with the red bits as the MSB's, the green bits next, and the blue bits as the LSB's.

Special note: As of this writing (Sep 88), Silver does NOT support anything but black for color zero.

Sampled sound format

IFF FORM "SAMP" Sampled Sound

Date: Dec 3,1989
From: Jim Fiore and Jeff Glatt, dissidents

The form "SAMP" is a file format used to store sampled sound data in some ways like the current standard, "8SVX". Unlike "8SVX", this new format is not restricted to 8 bit sample data. There can be more than one waveform per octave, and the lengths of different waveforms do not have to be factors of 2. In fact, the lengths (waveform size) and playback mapping (which musical notes each waveform will "play") are independently determined for each waveform. Furthermore, this format takes into account the MIDI sample dump standard (the defacto standard for musical sample storage), while also incorporating the ability to store Amiga specific info (for example, the sample data that might be sent to an audio channel which is modulating another channel).

Although this form can be used to store "sound effects" (typically oneShot sounds played at a set pitch), it is primarily intended to correct the many deficiencies of the "8SVX" form in regards to musical sampling. Because the emphasis is on musical sampling, this format relies on the MIDI (Musical Instrument Digital Interface) method of describing "sound events" as does virtually all currently manufactured, musical samplers. In addition, it attempts to incorporate features found on many professional music samplers, in anticipation that future Amiga models will implement 16 bit sampling, and thus be able to achieve this level of performance. Because this format is more complex than "8SVX", programming examples to demonstrate the use of this format have been included in both C and assembly. Also, a library of functions to read and write SAMP files is available, with example applications.

SEMANTICS: When MIDI literature talks about a sample, usually it means a collection of many sample points that make up what we call a "wave".

=====SIMILARITIES AND DIFFERENCES FROM THE "8SVX" FORM=======

Like "8SVX", this new format uses headers to separate the various sections of the sound file into chunks. Some of the chunks are exactly the same since there wasn't a need to improve them. The chunks that remain unchanged are as follows:

"(c) "
"AUTH"
"ANNO"

Since these properties are all described in the original "8SVX" document, please refer to that for a description of these chunks and their uses. Like the "8SVX" form, none of these chunks are required to be in a sound file. If they do appear, they must be padded out to an even number of bytes.

Furthermore, two "8SVX" chunks no longer exist as they have been incorporated into the "BODY" chunk. They are:

"ATAK"
"RLSE"

Since each wave can be completely different than the other waves in the sound file (one wave might be middle C on a piano, and another might be a snare drum hit), it is necessary for each wave to have its own envelope description, and name.

The major changes from the "8SVX" format are in the "MHDR", "NAME", and "BODY" chunks.

```
          =================THE "SAMP" HEADER================

At the very beginning of a sound file is the "SAMP" header. This is used to
determine if the disk file is indeed a SAMP sound file. It's attributes are
as follows:

#define ID_SAMP MakeID('S','A','M','P')

In assembly, this looks like:

      CNOP 0,2  ;word-align

SAMP         dc.b  'SAMP'
sizeOfChunks dc.l  [sizes of all subsequent chunks summed]


             =================THE "MHDR" CHUNK=================

  The required "MHDR" chunk immediately follows the "SAMP" header and consists
of the following components:

#define ID_MHDR MakeID('M','H','D','R')

  /* MHDR size is dependant on the size of the imbedded PlayMap. */

  typedef struct{
   UBYTE NumOfWaves,      /* The number of waves in this file */
     Format,              /* # of ORIGINAL significant bits from 8-28 */
     Flags,               /* Various bits indicate various functions */
     PlayMode,            /* determines play MODE of the PlayMap */
     NumOfChans,
     Pad,
     PlayMap[128*4],  /* a map of which wave numbers to use for
                          each of 128 possible Midi Notes. Default to 4 */
   } MHDRChunk;
```

The PlayMap is an array of bytes representing wave numbers. There can be a total of 255 waves in a "SAMP" file. They are numbered from 1 to 255. A wave number of 0 is reserved to indicate "NO WAVE". The Midi Spec 1.0 designates that there are 128 possible note numbers (pitches), 0 to 127. The size of an MHDR's PlayMap is determined by (NumOfChans * 128). For example, if NumOfChans = 4, then an MHDR's PlayMap is 512 bytes. There are 4 bytes in the PlayMap for EACH of the 128 Midi Note numbers. For example, the first 4 bytes in PlayMap pertain to Midi Note #0. Of those 4 bytes, the first byte is the wave number to play back on Amiga audio channel 0. The second byte is the wave number to play back on Amiga audio channel 1, etc. In this way, a single Midi Note Number could simultaneously trigger a sound event on each of the 4 Amiga audio channels. If NumOfChans is 1, then the PlayMap is 128 bytes and each midi note has only 1 byte in the PlayMap. The first byte pertains to midi note #0, the second pertains to midi note #1, etc. In this case, a player program might elect to simply play back the PlayMap wave number on any available amiga audio channel. If NumOfChans = 0, then there is no imbedded PlayMap in the MHDR, no midi note assignments for the waves, and an application should play back waves on any channel at their default sampleRates.

In effect, the purpose of the PlayMap array is to determine which (if any) waves are to be played back for each of the 128 possible Midi Note Numbers. Usually, the MHDR's NumOfChans will be set to 4 since the Amiga has 4 audio channels. For the rest of this document, the NumOfChans is assumed to be 4.

As mentioned, there can be a total of 255 waves in a "SAMP" file, numbered from 1 to 255. A PlayMap wave number of 0 is reserved to indicate that NO WAVE number should be played back. Consider the following example:

The first 4 bytes of PlayMap are 1,3,0,200.

If a sample playing program receives (from the serial port or another task perhaps) Midi Note Number 0, the following should occur:

```
  1) The sampler plays back wave 1 on Amiga audio channel
     number 0 (because the first PlayMap byte is 1).
  2) The sampler plays back wave 3 on Amiga audio channel
     number 1 (because the second PlayMap byte is 3).
  3) The sampler does not effect Amiga audio channel 2 in
     any way (because the third PlayMap byte is a 0).
  4) The sampler plays back wave 200 on Amiga audio channel
     number 4 (because the fourth PlayMap byte is 200).
```

(This assumes INDEPENDANT CHANNEL play MODE to be discussed later in this document.)

All four of the PlayMap bytes could even be the same wave number. This would cause that wave to be output of all 4 Amiga channels simultaneously.

NumOfWaves is simply the number of waves in the sound file.

Format is the number of significant bits in every sample of a wave. For example, if Format = 8, then this means that the sample data is an 8 bit format, and that every sample of the wave can be expressed by a single BYTE. (A 16 bit sample would need a WORD for every sample point).

Each bit of the Flags byte, when set, means the following:

```
Bit #0 - File continued on another disc. This might occur if the SAMP file
         was too large to fit on 1 floppy. The accepted practice (as incor-
         porated by Yamaha's TX sampler and Casio's FZ-1 for example) is to
         dump as much as possible onto one disc and set a flag to indicate
         that more is on another disc's file. The name of the files must
         be the related. The continuation file should have its own SAMP header
         MHDR, and BODY chunks. This file could even have its continuation
         bit set, etc. Never chop a sample wave in half. Always close the
         file on 1 disc after the last wave which can be completely saved.
         Resume with the next wave within the BODY of the continuation file.
         Also, the NumOfWaves in each file's BODY should be the number saved
         on that disc (not the total number in all combined disk files).
         See the end of this document for filename conventions.
```

In C, here is how the PlayMap is used when receiving a midi note-on event:

```
  MapOffset = (UBYTE) MidiNoteNumber * numOfChans;
  /* MidiNoteNumber is the received note number (i.e. the second byte of a
     midi note-on event. numOfChans is from the SAMP MHDR. */
  chan0waveNum = (UBYTE) playMap[MapOffset];
  chan1waveNum = (UBYTE) playMap[MapOffset+1];
  chan2waveNum = (UBYTE) playMap[MapOffset+2];
  chan3waveNum = (UBYTE) playMap[MapOffset+3];

  if (chan0waveNum != 0)
  { /* get the pointer to wave #1's data, determine the values
       that need to be passed to the audio device, and play this
       wave on Amiga audio channel #0 (if INDEPENDANT PlayMode) */
  }

   /* do the same with the other 3 channel's wave numbers */
```

In assembly, the "MHDR" structure looks like this:

```
              CNOP 0,2
MHDR         dc.b 'MHDR'
sizeOfMHDR   dc.l [this is 6 + (NumOfChans * 128) ]
NumOfWaves   dc.b [a byte count of the # of waves in the file]
Format       dc.b [a byte count of the # of significant bits in a sample point]
Flags        dc.b [bit mask]
PlayMode     dc.b [play MODE discussed later]
NumOfChans   dc.b [# of bytes per midi note for PlayMap]
PlayMap      ds.b [128 x NumOfChans bytes of initialized values]
```

and a received MidiNoteNumber is interpreted as follows:

```
   moveq    #0,d0
   move.b   MidiNoteNumber,d0   ;this is the received midi note #
   bmi.s    Illegal_Number      ;exit, as this is an illegal midi note #
   moveq    #0,d1
   move.b   NumOfChans,d1
   mulu.w   d1,d0               ;MidiNoteNumber x NumOfChans
   lea      PlayMap,a0
   adda.l   d0,a0
   move.b   (a0)+,chan0waveNum
   move.b   (a0)+,chan1waveNum
   move.b   (a0)+,chan2waveNum
   move.b   (a0),chan3waveNum

      tst.b   chan0waveNum
      beq.s   Chan1
 ;Now get the address of this wave number's sample data, determine the
 ;values that need to be passed to the audio device, and output the wave's
 ;data on Amiga chan 0 (assuming INDEPENDANT PlayMode).

Chan1 tst.b chan1waveNum
      beq.s Chan2
 ;do the same for the other wave numbers, etc.
```

====================THE "NAME" CHUNK========================

```
  #define ID_NAME MakeID('N','A','M','E')
```

If a NAME chunk is included in the file, then EVERY wave must have a name. Each name is NULL-terminated. The first name is for the first wave, and it is immediately followed by the second wave's name, etc. It is legal for a wave's name to be simply a NULL byte. For example, if a file contained 4 waves and a name chunk, the chunk might look like this:

```
           CNOP 0,2

Name       dc.b 'NAME'
sizeOfName dc.l 30
           dc.b 'Snare Drum',0  ;wave 1
           dc.b 'Piano 1',0     ;wave 2
           dc.b 'Piano A4',0    ;wave 3
           dc.b 0               ;wave 4
           dc.b 0
```

NAME chunks should ALWAYS be padded out to an even number of bytes. (Hence the extra NULL byte in this example). The chunk's size should ALWAYS be even consequently. DO NOT USE the typical IFF method of padding a chunk out to an even number of bytes, but allowing an odd number size in the header.

==============THE "BODY" CHUNK===============

The "BODY" chunk is CONSIDERABLY different than the "8SVX" form. Like all chunks it has an ID.

```
   #define ID_BODY MakeID('B','O','D','Y')
```

Every wave has an 80 byte waveHeader, followed by its data. The waveHeader structure is as follows:

```
 typedef struct {
   ULONG  WaveSize;        /* total # of BYTES in the wave (MUST be even) */
   UWORD  MidiSampNum;     /* ONLY USED for Midi Dumps */
   UBYTE  LoopType,        /* ONLY USED for Midi Dumps */
   InsType;            /* Used for searching for a certain instrument */
   ULONG  Period,       /* in nanoseconds at original pitch */
   Rate,                /* # of samples per second at original pitch */
   LoopStart,            /* an offset in BYTES (from the beginning of the
                             of the wave) where the looping portion of the
                             wave begins. Set to WaveSize if no loop. */
   LoopEnd;              /* an offset in BYTES (from the beginning of the
                             of the wave) where the looping portion of the
                             wave ends. Set to WaveSize if no loop. */
   UBYTE  RootNote,        /* the Midi Note # that plays back original pitch */
   VelStart;            /* 0 = NO velocity effect, 128 =
                               negative direction, 64 = positive
                               direction (it must be one of these 3) */
   UWORD VelTable[16];  /* contains 16 successive offset values
                               in BYTES from the beginning of the wave */

  /* The ATAK and RLSE segments contain an EGPoint[] piece-wise
     linear envelope just like 8SVX. The structure of an EGPoint[]
     is the same as 8SVX. See that document for details. */

   ULONG  ATAKsize,     /* # of BYTES in subsequent ATAK envelope.
                            If 0, then no ATAK data for this wave. */
   RLSEsize,            /* # of BYTES in subsequent RLSE envelope
                            If 0, then no RLSE envelope follows */

  /* The FATK and FRLS segments contain an EGPoint[] piece-wise
     linear envelope for filtering purposes. This is included in
     the hope that future Amiga audio will incorporate a VCF
     (Voltage Controlled Filter). Until then, if you are doing any
     non-realtime digital filtering, you could store info here. */

  sizeOfFATK,            /* # of BYTES in FATK segment */
  sizeOfFRLS,            /* # of BYTES in FRLS segment */

  USERsize;        /*   # of BYTES in the following data
                             segment (not including USERtype).
                             If zero, then no user data */
  UWORD  USERtype;     /* See explanation below. If USERsize
                            = 0, then ignore this. */

 /* End of the waveHeader. */

 /* The data for any ATAK, RLSE, FATK, FRLS, USER, and the actual wave
    data for wave #1 follows in this order:
    Now list each EGPoint[] (if any) for the VCA's (Voltage Controlled Amp)
    attack portion.
    Now list each EGPoint[] for the VCA's (Voltage Controlled Amp)
    release portion.
    List EGPoints[] (if any) for FATK.
    List EGPoints[] if any for FRLS */
```

```
/*  Now include the user data here if there is any. Just pad it out
    to an even number of bytes and have USERsize reflect that.
    Finally, here is the actual sample data for the wave. The size (in BYTES)
    of this data is WaveSize. It MUST be padded out to an even number of bytes. */

 } WaveFormInfo;

 /* END OF WAVE #1 */

 /* The waveHeader and data for the next wave would now follow. It is
    the same form as the first wave */


In assembly,  the BODY chunk looks like this:

           CNOP 0,2
BodyHEADER dc.b 'BODY'
sizeOfBody dc.l  [total bytes in the BODY chunk not counting 8 byte header]

   ; Now for the first wave
WaveSize        dc.l  ;[total # of BYTES in this wave (MUST be even)]
MidiSampNum     dc.w  ;[from Midi Sample Dump]  ; ONLY USED for Midi Dumps
LoopType        dc.b  ;[0 or 1]                 ; ONLY USED for Midi Dumps
InsType         dc.b  0
Period          dc.l  ;[period in nanoseconds at original pitch]
Rate            dc.l  ;[# of samples per second at original pitch]
LoopStart       dc.l    ;[an offset in BYTES (from the beginning of the
                        ; of the wave) to where the looping
                        ; portion of the wave begins.]
LoopEnd         dc.l    ;[an offset in BYTES (from the beginning of the
                        ; of the wave) to where the looping
                        ; portion of the wave ends]
RootNote        dc.b    ;[the Midi Note # that plays back original pitch]
VelStart        dc.b    ;[0, 64, or 128]
VelTable        dc.w    ;[first velocity offset]
                dc.w    ;[second velocity offset]...etc
                ds.w 14 ;...for a TOTAL of 16 velocity offsets

ATAKsize        dc.l  ;# of BYTES in subsequent ATAK envelope.
                      ;If 0, then no ATAK data for this wave.
RLSEsize        dc.l  ;# of BYTES in subsequent RLSE envelope
                      ;If 0, then no RLSE data
FATKsize        dc.l  ;# of BYTES in FATK segment
FRLSsize        dc.l  ;# of BYTES in FRLS segment
USERsize        dc.l  ;# of BYTES in the following User data
                      ;segment (not including USERtype).
                      ;If zero, then no user data
USERtype        dc.w  ; See explanation below. If USERsize
                   ; = 0, then ignore this.

  ;Now include the EGpoints[] (data) for the ATAK if any
  ;Now the EGpoints for the RLSE
  ;Now the EGpoints for the FATK
  ;Now the EGpoints for the FLSR
  ;Now include the user data here if there is any. Just pad
  ;it out to an even number of bytes.
  ;After the userdata (if any) is the actual sample data for
  ;the wave. The size (in BYTES) of this segment is WaveSize.
  ;It MUST be padded out to an even number of bytes.

  ; END OF WAVE #1
```

=============STRUCTURE OF AN INDIVIDUAL SAMPLE POINT=============

Even though the next generation of computers will probably have 16 bit audio, and 8 bit sampling will quickly disappear, this spec has sizes expressed in BYTES. (ie LoopStart, WaveSize, etc.) This is because each successive address in RAM is a byte to the 68000, and so calculating address offsets will be much easier with all sizes in BYTES. The Midi sample dump, on the other hand, has sizes expressed in WORDS. What this means is that if you have a 16 bit wave, for example, the WaveSize is the total number of BYTES, not WORDS, in the wave.

Also, there is no facility for storing a compression type. This is because sample data should be stored in linear format (as per the MIDI spec). Currently, all music samplers, regardless of their internal method of playing sample data must transmit and expect to receive sample dumps in a linear format. It is up to each device to translate the linear format into its own compression scheme. For example, if you are using an 8 bit compression scheme that yields a 14 bit linear range, you should convert each sample data BYTE to a decompressed linear WORD when you save a sound file. Set the MHDR's Format to 14. It is up to the application to do its own compression upon loading a file. The midi spec was set up this way because musical samplers need to pass sample data between each other, and computers (via a midi interface). Since there are almost as many data compression schemes on the market as there are musical products, it was decided that all samplers should expect data received over midi to be in LINEAR format. It seems logical to store it this way on disc as well. Therefore, any software program "need not know" how to decompress another software program's SAMP file. When 16 bit sampling is eventually implemented there won't be much need for compression on playback anyway. The continuation Flag solves the problem of disc storage as well.

Since the 68000 can only perform math on BYTES, WORDS, or LONGS, it has been decided that a sample point should be converted to one of these sizes when saved in SAMP as follows:

```
 ORIGINAL significant bits          SAMP sample point size
M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--
  M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--M--
           8                                BYTE
         9 to 16                            WORD
         17 to 28                           LONG
```

Furthermore, the significant bits should be left-justified since it is easier to perform math on the samples.

So, for example, an 8 bit sample point (like 8SVX) would be saved as a BYTE with all 8 bits being significant. The MHDR's Format = 8. No conversion is necessary.

A 12 bit sample point should be stored as a WORD with the significant bits being numbers 4 to 15. (i.e shift the 12-bit WORD 4 places to the left). Bits 0, 1, 2 and 3 may be zero (unless some 16-bit math was performed and you wish to save these results). The MHDR's Format = 12. In this way, the sample may be loaded and manipulated as a 16-bit wave, but when transmitted via midi, it can be converted back to 12 bits (rounded and shifted right by 4).

A 16 bit sample point would be saved as a WORD with all 16 bits being significant. The MHDR's Format = 16. No conversion is necessary.

============== The waveHeader explained ==============

The WaveSize is, as stated, the number of BYTES in the wave's sample table. If your sample data consisted of the following 8 bit samples:

```
    BYTE  100,-90,80,-60,30,35,40,-30,-35,-40,00,12,12,10
```

then WaveSize = 14. (PAD THE DATA OUT TO AN EVEN NUMBER OF BYTES!)

The MidiSampNum is ONLY used to hold the sample number received from a MIDI Sample Dump. It has no bearing on where the wave should be placed in a SAMP file. Also, the wave numbers in the PlayMap are between 1 to 255, with 1 being the number of the first wave in the file. Remember that a wave number of 0 is reserved to mean "no wave to play back". Likewise, the LoopType is only used to hold info from a MIDI sample dump.

The InsType is explained at the end of this document. Often it will be set to 0.

The RootNote is the Midi Note number that will play the wave back at it's original, recorded pitch. For example, consider the following excerpt of a PlayMap:

```
  PlayMap  {2,0,0,4        /* Midi Note #0 channel assignment */
            4,100,1,0      /* Midi Note #1    "         "  */
            1,4,0,0        /* Midi Note #2    "         "  */
            60,2,1,1...}   /* Midi Note #3    "         "  */
```

Notice that Midi Notes 0, 1, and 2 are all set to play wave number 4 (on Amiga channels 3, 0, and 1 respectively). If we set wave 4's RootNote = 1, then receiving Midi Note number 1 would play back wave 4 (on Amiga channel 0) at it's original pitch. If we receive a Midi Note number 0, then wave 4 would be played back on channel 3) a half step lower than it's original pitch. If we receive Midi Note number 2, then wave 4 would be played (on channel 1) a half step higher than it's original pitch. If we receive Midi Note number 3, then wave 4 would not be played at all because it isn't specified in the PlayMap bytes for Midi Note number 3.

The Rate is the number of samples per second of the original pitch. For example, if Rate = 20000, then to play the wave at it's original pitch, the sampling period would be:

```
     (1/20000)/.279365 = .000178977
```

```
#define AUDIO_HARDWARE_FUDGE .279365
```

where .279365 is the Amiga Fudge Factor (a hardware limitation). Since the Amiga needs to see the period in terms of microseconds, move the decimal place to the right 6 places and our sampling period = 179 (rounded to an integer).

In order to play the wave at higher or lower pitches, one would need to "transpose" this period value. By specifying a higher period value, the Amiga will play back the samples slower, and a lower pitch will be achieved. By specifying a lower period value, the amiga will play back the sample faster, and a higher pitch will be achieved. By specifying this exact period, the wave will be played back exactly "as it was recorded (sampled)". ("This period is JUST RIGHT!", exclaimed GoldiLocks.) Later, a method of transposing pitch will be shown using a "look up" table of periods. This should prove to be the fastest way to transpose pitch, though there is nothing in the SAMP format that compels you to do it this way.

The LoopStart is a BYTE offset from the beginning of the wave to where the looping portion of the wave begins. For example, if SampleData points to the start of the wave, then SampleData + LoopStart is the start address of the looping portion. In 8SVX, the looping portion was referred to as repeatHiSamples. The data from the start of the wave up to the start of the looping portion is the oneShot portion of the wave. LoopEnd is a BYTE offset from the beginning of the wave to where the looping portion ends. This might be the very end of the wave in memory, or perhaps there might be still more data after this point. You can choose to ignore this "trailing" data and play back the two other portions of the wave just like an 8SVX file (except that there are no other interpolated octaves of this wave).

VelTable contains 16 BYTE offsets from the beginning of the wave. Each successive value should be greater (or equal to) the preceding value. If VelStart = POSITIVE (64), then for each 8 increments in Midi Velocity above 0, you move UP in the table, add this offset to the wave's beginning address (start of oneShot), and start playback at that address. Here is a table relating received midi note-on velocity vs. start playback address for POSITIVE VelStart. SamplePtr points to the beginning of the sample.

```
 If midi velocity = 0, then don't play a sample, this is a note off
 If midi velocity = 1 to 7, then start play at SamplePtr + VelTable[0]
 If midi velocity = 8 to 15, then start at SamplePtr + VelTable[1]
 If midi velocity = 16 to 23, then start at SamplePtr + VelTable[2]
 If midi velocity = 24 to 31, then start at SamplePtr + VelTable[3]
 If midi velocity = 32 to 39, then start at SamplePtr + VelTable[4]
 If midi velocity = 40 to 47, then start at SamplePtr + VelTable[5]
 If midi velocity = 48 to 55, then start at SamplePtr + VelTable[6]
 If midi velocity = 56 to 63, then start at SamplePtr + VelTable[7]
 If midi velocity = 64 to 71, then start at SamplePtr + VelTable[8]
 If midi velocity = 72 to 79, then start at SamplePtr + VelTable[9]
 If midi velocity = 80 to 87, then start at SamplePtr + VelTable[10]
 If midi velocity = 88 to 95, then start at SamplePtr + VelTable[11]
 If midi velocity = 96 to 103, then start at SamplePtr + VelTable[12]
 If midi velocity = 104 to 111, then start at SamplePtr + VelTable[13]
 If midi velocity = 112 to 119, then start at SamplePtr + VelTable[14]
 If midi velocity = 120 to 127, then start at SamplePtr + VelTable[15]
```

We don't want to specify a scale factor and use integer division to find the sample start. This would not only be slow, but also, it could never be certain that the resulting sample would be a zero crossing if the start point is calculated "on the fly". The reason for having a table is so that the offsets can be be initially set on zero crossings via an editor. This way, no audio "clicks" is guaranteed. This table should provide enough resolution.

If VelStart = NEGATIVE (128), then for each 8 increments in midi velocity, you start from the END of VelTable, and work backwards. Here is a table for NEGATIVE velocity start.

```
 If midi velocity = 0, then don't play a sample, this is a note off
 If midi velocity = 1 to 7, then start play at SamplePtr + VelTable[15]
 If midi velocity = 8 to 15, then start at SamplePtr + VelTable[14]
 If midi velocity = 16 to 23, then start at SamplePtr + VelTable[13]
 If midi velocity = 24 to 31, then start at SamplePtr + VelTable[12]
 If midi velocity = 32 to 39, then start at SamplePtr + VelTable[11]
 If midi velocity = 40 to 47, then start at SamplePtr + VelTable[10]
 If midi velocity = 48 to 55, then start at SamplePtr + VelTable[9]
 If midi velocity = 56 to 63, then start at SamplePtr + VelTable[8]
 If midi velocity = 64 to 71, then start at SamplePtr + VelTable[7]
 If midi velocity = 72 to 81, then start at SamplePtr + VelTable[6]
 If midi velocity = 80 to 87, then start at SamplePtr + VelTable[5]
 If midi velocity = 88 to 95, then start at SamplePtr + VelTable[4]
 If midi velocity = 96 to 103, then start at SamplePtr + VelTable[3]
 If midi velocity = 104 to 111, then start at SamplePtr + VelTable[2]
 If midi velocity = 112 to 119, then start at SamplePtr + VelTable[1]
 If midi velocity = 120 to 127, then start at SamplePtr + VelTable[0]
```

In essence, increasing midi velocity starts playback "farther into" the wave for POSITIVE VelStart. Increasing midi velocity "brings the start point back" toward the beginning of the wave for NEGATIVE VelStart.

If VelStart is set to NONE (0), then the wave's playback start should not be affected by the table of offsets.

What is the use of this feature? As an example, when a snare drum is hit with a soft volume, its initial attack is less pronounced than when it is struck hard. You might record a snare being hit hard. By setting VelStart to a NEGATIVE value and setting up the offsets in the Table, a lower midi velocity will "skip" the beginning samples and thereby tend to soften the initial attack. In this way, one wave yields a true representation of its instrument throughout its volume range. Furthermore, stringed and plucked instruments (violins, guitars, pianos, etc) exhibit different attacks at different volumes. VelStart makes these kinds of waves more realistic via a software implementation. Also, an application program can allow the user to enable/disable this feature. See the section "Making the Velocity Table" for info on how to best choose the 16 table values.

=========MIDI VELOCITY vs. AMIGA CHANNEL VOLUME============

The legal range for Midi Velocity bytes is 0 to 127. (A midi velocity of 0 should ALWAYS be interpreted as a note off).

The legal range for Amiga channel volume is 0 to 64. Since this is half of the midi range, a received midi velocity should be divided by 2 and add 1 (but only AFTER checking for a received midi velocity of 0).

An example of how to implement a received midi velocity in C:

```
If ( ReceivedVelocity != 0 && ReceivedVelocity < 128 )
{   /* the velocity byte of a midi message */
    If (velStart != 0)
    {
        tableEntry = ReceivedVelocity / 8;
        If (velStart == 64)
        {    /* Is it POSITIVE */
             startOfWave = SamplePtr + velTable[tableEntry];
                         /* ^where to find the sample start point */
        }
        If (velStart == 128)
        {    /* Is it NEGATIVE */
             startOfWave = SamplePtr + velTable[15 - tableEntry];
        }
        volume = (receivedVelocity/2 + 1;  /* playback volume */
        /* Now playback the wave */
    }
}
```

In assembly,

```
  lea      SampleData,a0        ;the start addr of the sample data
  moveq    #0,d0
  move.b   ReceivedVelocity,d0  ;the velocity byte of a midi message
  beq      A_NoteOff            ;If zero, branch to a routine to
                                ;process a note-off message.

  bmi      Illegal_Vol          ;exit if received velocity > 127
  ;---Check for velocity start feature ON, and direction
  move.b   VelStart,d1
  beq.s    Volume               ;skip the velocity offset routine if 0
  bmi.s    NegativeVel          ;is it NEGATIVE? (128)

  ;---Positive velocity offset
  move.l   d0,d1                ;duplicate velocity
  lsr.b    #3,d1                ;divide by 8
  add.b    d1,d1                ;x 2 because we need to fetch a word
  lea      VelTable,a1       ;start at table's HEAD
  adda.l   d1,a1                ;go forward
  move.w   (a1),d1              ;get the velocity offet
  adda.l   d1,a0           ;where to start actual playback
  bra.s    Volume

NegativeVel:
  ;---Negative velocity offset
  move.l   d0,d1                ;duplicate velocity
  lsr.b    #3,d1                ;divide by 8
  add.b    d1,d1                ;x 2 because we need to fetch a word
  lea      VelTable+30,a1   ;start at table's END
  suba.l   d1,a1                ;go backwards
  move.w   (a1),d1              ;get the velocity offset
  adda.l   d1,a0           ;where to start actual playback

  ;---Convert Midi velocity to an Amiga volume
Volume  lsr.b    #1,d0          ;divide by 2
        addq.b   #1,d0          ;an equivalent Amiga volume

 ;---Now a0 and d0 are the address of sample start, and volume
```

================= AN EGpoint (envelope generator) ================

A single EGpoint is a 6 byte structure as follows:

```
EGpoint1: dc.w ;[the duration in milliseconds]
          dc.l ;[the volume factor - fixed point, 16 bits to the left of the
               ;decimal point and 16 to the right.]
```

The volume factor is a fixed point where 1.0 ($00010000) represents the MAXIMUM volume possible. (i.e. No volume factor should exceed this value.) The last EGpoint in the ATAK is always the sustain point. Each EG's volume is determined from 0.0, not as a difference from the previous EG's volume. I hope that this clears up the ambiguity in the original 8SVX document. So, to recreate an amplifier envelope like this:

```
    /\
   /  \____
  /        \
 /          \

 |  | |    |  |
  1  2  3    4
```

Stages 1, 2, and 3 would be in the ATAK data, like so:

```
;Stage 1
dc.w  100        ;take 100ms
dc.l  $00004000 ;go to this volume
dc.w  100
dc.l  $00008000
dc.w  100
dc.l  $0000C000
dc.w  100
dc.l  $00010000 ;the "peak" of our attack is full volume
;Stage 2
dc.w  100
dc.l  $0000C000 ;back off to this level
dc.l  100
dc.l  $00008000 ;this is where we hold (SUSTAIN) until the note is turned
                ;off. (We are now holding at stage 3)
```

Now the RLSE data would specify stage 4 as follows:

```
dc.w  100
dc.l  $00004000
dc.w  100
dc.l  $00000000 ;the volume is 0
```

===============ADDITIONAL USER DATA SECTION=================

There is a provision for storing user data for each wave. This is where an application can store Amiga hardware info, or other, application specific info. The waveHeader's USERtype tells what kind of data is stored. The current types are:

```
#define SPECIFIC 0
#define VOLMOD   1
#define PERMOD   2
#define LOOPING  3
```

SPECIFIC (0) - application specific data. It should be stored in a format that some application can immediately recognize. (i.e. a "format within" the SAMP format) If the USERtype is SPECIFIC, and an application doesn't find some sort of header that it can re-cognize, it should conclude that this data was put there by "someone else", and ignore the data.

VOLMOD (1) - This data is for volume modulation of an Amiga channel as described by the ADKCON register. This data will be sent to the modulator channel of the channel set to play the wave.

PERMOD (2) - This data is for period modulation of an Amiga channel as described by the ADKCON register. This data will be sent to the modulator channel of the channel set to play the wave.

LOOPING (3) - This contains more looping points for the sample. There are some samplers that allow more than just one loop (Casio products primarily). Additional looping info can be stored in this format:

```
UWORD numOfLoops;  /* number of loop points to follow */

ULONG StartLoop1,  /* BYTE offset from the beginning of
                      the sample to the start of loop1 */
EndLoop1,          /* BYTE offset from the beginning of
                      the sample to the end of loop1 */

StartLoop2,        /* ...etc */
```

=========Converting Midi Sample Dump to SAMP=========

SEMANTICS: When MIDI literature talks about a sample, usually it means a collection of many sample points that make up what we call "a wave". Therefore, a Midi Sample Dump sends all the sample data that makes up ONE wave. A SAMP file is designed to hold up to 255 of these waves (midi dumps).

The Midi Sample Dump specifies playback rate only in terms of a sample PERIOD in nanoseconds. SAMP also expresses playback in terms of samples per second (frequency). The Amiga needs to see its period rounded to the nearest microsecond. If you take the sample period field of a Midi sample Dump (the 8th, 9th, and 10th bytes of the Dump Header LSB first) which we will call MidiSamplePer, and the Rate of a SAMP file, here is the relationship:

```
Rate = (1/MidiSamplePer) x 10E9
```

Also the number of samples (wave's length) in a Midi Sample Dump (the 11th, 12th, and 13th bytes of the Dump header) is expressed in WORDS. SAMP's WaveSize is expressed in the number of BYTES. (For the incredibly stupid), the relationship is:

```
WaveSize = MidiSampleLength x 2
```

A Midi sample dump's LoopStart point and LoopEnd point are also in WORDS as versus the SAMP equivalents expressed in BYTES.

A Midi sample dump's sample number can be 0 to 65535. A SAMP file can hold up to 255 waves, and their numbers in the playmap must be 1 to 255. (A single, Midi Sample Dump only sends info on one wave.) When recieving a Midi Sample Dump, just store the sample number (5th and 6th bytes of the Dump Header LSB first) in SAMP's MidiSampNum field. Then forget about this number until you need to send the wave back to the Midi instrument from whence it came.

A Midi Dump's loop type can be forward, or forward/backward. Amiga hardware supports forward only. You should store the Midi Dump's LoopType byte here, but ignore it otherwise until/unless Amiga hardware supports "reading audio data" in various ways. If so, then the looptype is as follows:

```
forward = 0, backward/forward = 1
```

A Midi Dump's sample format byte is the same as SAMP's.

===================== INTERPRETING THE PLAYMODE ==========================

PlayMode specifies how the bytes in the PlayMap are to be interpreted. Remember that a PlayMap byte of 0 means "No Wave to Play".

```
#define INDEPENDANT 0
#define MULTI       1
#define STEREO      2
#define PAN         3
```

PlayMode types:

```
 INDEPENDANT  (0) - The wave #s for a midi note are to be output on
                    Amiga audio channels 0, 1, 2, and 3 respectively.
                    If the NumOfChans is < 4, then only use that many channels.

 MULTI        (1) - The first wave # (first of the PlayMap bytes) for a
                    midi note is to be output on any free channel. The other
                    wave numbers are ignored. If all four channels are in
                    play, the application can decide whether to "steal" a
                    channel.

 STEREO      (2) -  The first wave # (first of the PlayMap bytes) is to be
                    output of the Left stereo jack (channel 1 or 3) and if
                    there is a second wave number (the second of the PlayMap
                    bytes), it is to be output the Right jack (channel 2 or 4).
                    The other wave numbers are ignored.

 PAN         (3) -  This is just like STEREO except that the volume of wave 1
                    should start at its initial volume (midi velocity) and
                    fade to 0. At the same rate, wave 2 should start at 0
                    volume and rise to wave #1's initial level. The net
                    effect is that the waves "cross" from Left to Right in
                    the stereo field. This is most effective when the wave
                    numbers are the same. (ie the same wave) The application
                    program should set the rate. Also, the application can
                    reverse the stereo direction (ie Right to Left fade).
```

The most important wave # to be played back by a midi note should be the first of the PlayMap bytes. If the NumOfChans > 1, the second PlayMap byte should be a defined wave number as well (even if it is deliberately set to the same value as the first byte). This insures that all 4 PlayModes will have some effect on a given SAMP file. Also, an application should allow the user to change the PlayMode at will. The PlayMode stored in the SAMP file is only a default or initial set-up condition.

=================== MAKING A TRANSPOSE TABLE =====================

In order to allow a wave to playback over a range of musical notes, (+/- semitones), its playback rate must be raised or lowered by a set amount. From one semitone to the next, this set amount is by a factor of the 12th root of 2 (assuming a western, equal-tempered scale). Here is a table that shows what factor would need to be multiplied by the sampling rate in order to transpose the wave's pitch.

Pitch in relation to the Root	Multiply Rate by this amount	
DOWN 6 semitones	0.5	
DOWN 5 1/2 semitones	0.529731547	
DOWN 5 semitones	0.561231024	
DOWN 4 1/2 semitones	0.594603557	
DOWN 4 semitones	0.629960525	
DOWN 3 1/2 semitones	0.667419927	
DOWN 3 semitones	0.707106781	
DOWN 2 1/2 semitones	0.749153538	
DOWN 2 semitones	0.793700526	
DOWN 1 1/2 semitones	0.840896415	
DOWN 1 semitones	0.890898718	
DOWN 1/2 semitone	0.943874312	
ORIGINAL_PITCH	1.0	/* rootnote's pitch */
UP 1/2 semitone	1.059463094	
UP 1 semitones	1.122562048	
UP 1 1/2 semitones	1.189207115	
UP 2 semitones	1.259921050	
UP 2 1/2 semitones	1.334839854	
UP 3 semitones	1.414213562	
UP 3 1/2 semitones	1.498307077	
UP 4 semitones	1.587401052	
UP 4 1/2 semitones	1.681792830	
UP 5 semitones	1.781797436	
UP 5 1/2 semitones	1.887748625	
UP 6 semitones	2	

For example, if the wave's Rate is 18000 hz, and you wish to play the wave UP 1 semitone, then the playback rate is:

18000 x 1.122562048 = 20206.11686 hz

The sampling period for the Amiga is therefore:

(1/20206.11686)/.279365 = .000177151

and to send it to the Audio Device, it is rounded and expressed in micro-seconds: 177

Obviously, this involves floating point math which can be time consuming and impractical for outputing sound in real-time. A better method is to construct a transpose table that contains the actual periods already calculated for every semitone. The drawback of this method is that you need a table for EVERY DIFFERENT Rate in the SAMP file. If all the Rates in the file happened to be the same, then only one table would be needed. Let's assume that this is the case, and that the Rate = 18000 hz. Here is a table containing enough entries to transpose the waves +/- 6 semitones.

```
 Pitch in relation to the Root     The Amiga Period (assuming rate = 18000 hz)
 ---------------------------------  ------------------------------
Transposition_table[TRANS_TABLE_SIZE]={
/* DOWN 6     semitones  */           398,
/* DOWN 5 1/2 semitones  */           375,
/* DOWN 5     semitones  */           354,
/* DOWN 4 1/2 semitones  */           334,
/* DOWN 4     semitones  */           316,
/* DOWN 3 1/2 semitones  */           298,
/* DOWN 3     semitones  */           281,
/* DOWN 2 1/2 semitones  */           265,
/* DOWN 2     semitones  */           251,
/* DOWN 1 1/2 semitones  */           236,
/* DOWN 1     semitones  */           223,
/* DOWN 1/2   semitone   */           211,
/* ORIGINAL_PITCH        */           199,      /* rootnote's pitch */
/* UP   1/2   semitone   */            187,
/* UP   1     semitones  */            177,
/* UP   1 1/2 semitones  */            167,
/* UP   2     semitones  */            157,
/* UP   2 1/2 semitones  */            148,
/* UP   3     semitones  */            141,
/* UP   3 1/2 semitones  */            133,
 /* Since the minimum Amiga period = 127 the following
    are actually out of range. */
/* UP   4     semitones  */            125,
/* UP   4 1/2 semitones  */            118,
/* UP   5     semitones  */            112,
/* UP   5 1/2 semitones  */            105,
/* UP   6     semitones  */            99   };
```

Let's assume that (according to the PlayMap) midi note #40 is set to play wave number 3. Upon examining wave 3's structure, we discover that the Rate = 18000, and the RootNote = 38. Here is how the Amiga sampling period is calulated using the above 18000hz "transpose chart" in C:

```
  /* MidiNoteNumber is the received midi note's number (here 40) */

  #define ORIGINAL_PITCH     TRANS_TABLE_SIZE/2 + 1
/* TRANS_TABLE_SIZE is the number of entries in the transposition table
   (dynamic, ie this can change with the application) */

  transposeAmount = (LONG) (MidiNoteNumber - rootNote); /* make it a SIGNED LONG */
  amigaPeriod     = Transposition_table[ORIGINAL_PITCH + transposeAmount];
```

In assembly, the 18000hz transpose chart and above example would be:

```
Table         dc.w  398
              dc.w  375
              dc.w  354
              dc.w  334
              dc.w  316
              dc.w  298
              dc.w  281
              dc.w  265
              dc.w  251
              dc.w  236
              dc.w  223
              dc.w  211
ORIGINAL_PITCH  dc.w  199   ; rootnote's pitch
              dc.w  187
              dc.w  177
              dc.w  167
              dc.w  157
              dc.w  148
              dc.w  141
              dc.w  133
```

```
 ; Since the minimum Amiga period = 127, the following
 ; are actually out of range.
            dc.w  125
            dc.w  118
            dc.w  112
            dc.w  105
            dc.w  99

  lea      ORIGINAL_PITCH,a0
  move.b   MidiNoteNumber,d0   ;the received note number
  sub.b    RootNote,d0         ;subtract the wave's root note
  ext.w    d0
  ext.l    d0                  ;make it a signed LONG
  add.l    d0,d0               ;x 2 in order to fetch a WORD
  adda.l   d0,a0
  move.w   (a0),d0             ;the Amiga Period (WORD)
```

Note that these examples don't check to see if the transpose amount is beyond the number of entries in the transpose table. Nor do they check if the periods in the table are out of range of the Amiga hardware.

===================== MAKING THE VELOCITY TABLE ======================

The 16 entries in the velocity table should be within the oneShot portion of the sample (ie not in the looping portion). THe first offset, VelTable[0] should be set to zero (in order to play back from the beginning of the data). The subsequent values should be increasing numbers. If you are using a graphic editor, try choosing offsets that will keep you within the initial attack portion of the wave. In practice, these values will be relatively close together within the wave. Always set the offsets so that when they are added to the sample start point, the resulting address points to a sample value of zero (a zero crossing point). This will eliminate pops and clicks at the beginning of the playback.

In addition, the start of the wave should be on a sample with a value of zero. The last sample of the oneShot portion and the first sample of the looping portion should be approximately equal, (or zero points). The same is true of the first and last samples of the looping portion. Finally, try to keep the slopes of the end of the oneShot, the beginning of the looping, and the end of the looping section, approximately equal. All this will eliminate noise on the audio output and provide "seamless" looping.

======================== THE INSTRUMENT TYPE ==========================

Many SMUS players search for certain instruments by name. Not only is this slow (comparing strings), but if the exact name can't be found, then it is very difficult and time-consuming to search for a suitable replacement. For this reason, many SMUS players resort to "default" instruments even if these are nothing like the desired instruments. The InsType byte in each waveHeader is meant to be a numeric code which will tell an SMUS player exactly what the instrument is. In this way, the SMUS player can search for the correct "type" of instrument if it can't find the desired name. The type byte is divided into 2 nibbles (4 bits for you C programmers) with the low 4 bits representing the instrument "family" as follows:

```
 1 = STRING, 2 = WOODWIND, 3 = KEYBOARD, 4 = GUITAR, 5 = VOICE, 6 = DRUM1,
 7 = DRUM2,  8 = PERCUSSION1, 9 = BRASS1, A = BRASS2, B = CYMBAL, C = EFFECT1,
 D = EFFECT2, E = SYNTH, F is undefined at this time
```

Now, the high nibble describes the particular type within that family.

For the STRING family, the high nibble is as follows:

1 = VIOLIN BOW, 2 = VIOLIN PLUCK, 3 = VIOLIN GLISSANDO, 4 = VIOLIN TREMULO, 5 = VIOLA BOW, 6 = VIOLA PLUCK, 7 = VIOLA GLIS, 8 = VIOLA TREM, 9 = CELLO BOW, A = CELLO PLUCK, B = CELLO GLIS, C = CELLO TREM, D = BASS BOW, E = BASS PLUCK (jazz bass), F = BASS TREM

For the BRASS1 family, the high nibble is as follows:

1 = BARITONE SAX, 2 = BARI GROWL, 3 = TENOR SAX, 4 = TENOR GROWL, 5 = ALTO SAX, 6 = ALTO GROWL, 7 = SOPRANO SAX, 8 = SOPRANO GROWL, 9 = TRUMPET, A = MUTED TRUMPET, B = TRUMPET DROP, C = TROMBONE, D = TROMBONE SLIDE, E = TROMBONE MUTE

For the BRASS2 family, the high nibble is as follows:

1 = FRENCH HORN, 2 = TUBA, 3 = FLUGAL HORN, 4 = ENGLISH HORN

For the WOODWIND family, the high nibble is as follows:

1 = CLARINET, 2 = FLUTE, 3 = PAN FLUTE, 4 = OBOE, 5 = PICCOLO, 6 = RECORDER, 7 = BASSOON, 8 = BASS CLARINET, 9 = HARMONICA

For the KEYBOARD family, the high nibble is as follows:

1 = GRAND PIANO, 2 = ELEC. PIANO, 3 = HONKYTONK PIANO, 4 = TOY PIANO, 5 = HARPSICHORD, 6 = CLAVINET, 7 = PIPE ORGAN, 8 = HAMMOND B-3, 9 = FARFISA ORGAN, A = HARP

For the DRUM1 family, the high nibble is as follows:

1 = KICK, 2 = SNARE, 3 = TOM, 4 = TIMBALES, 5 = CONGA HIT, 6 = CONGA SLAP, 7 = BRUSH SNARE, 8 = ELEC SNARE, 9 = ELEC KICK, A = ELEC TOM, B = RIMSHOT, C = CROSS STICK, D = BONGO, E = STEEL DRUM, F = DOUBLE TOM

For the DRUM2 family, the high nibble is as follows:

1 = TIMPANI, 2 = TIMPANI ROLL, 3 = LOG DRUM

For the PERCUSSION1 family, the high nibble is as follows:

1 = BLOCK, 2 = COWBELL, 3 = TRIANGLE, 4 = TAMBOURINE, 5 = WHISTLE, 6 = MARACAS, 7 = BELL, 8 = VIBES, 9 = MARIMBA, A = XYLOPHONE, B = TUBULAR BELLS, C = GLOCKENSPEIL

For the CYMBAL family, the high nibble is as follows:

1 = CLOSED HIHAT, 2 = OPEN HIHAT, 3 = STEP HIHAT, 4 = RIDE, 5 = BELL CYMBAL, 6 = CRASH, 7 = CHOKE CRASH, 8 = GONG, 9 = BELL TREE, A = CYMBAL ROLL

For the GUITAR family, the high nibble is as follows:

1 = ELECTRIC, 2 = MUTED ELECTRIC, 3 = DISTORTED, 4 = ACOUSTIC, 5 = 12-STRING, 6 = NYLON STRING, 7 = POWER CHORD, 8 = HARMONICS, 9 = CHORD STRUM, A = BANJO, B = ELEC. BASS, C = SLAPPED BASS, D = POPPED BASS, E = SITAR, F = MANDOLIN (Note that an acoustic picked bass is found in the STRINGS - Bass Pluck)

For the VOICE family, the high nibble is as follows:

1 = MALE AHH, 2 = FEMALE AHH, 3 = MALE OOO, 4 = FEMALE OOO, 5 = FEMALE BREATHY, 6 = LAUGH, 7 = WHISTLE

For the EFFECTS1 family, the high nibble is as follows:

1 = EXPLOSION, 2 = GUNSHOT, 3 = CREAKING DOOR OPEN, 4 = DOOR SLAM, 5 = DOOR CLOSE, 6 = SPACEGUN, 7 = JET ENGINE, 8 = PROPELLER, 9 = HELOCOPTER, A = BROKEN GLASS, B = THUNDER, C = RAIN, D = BIRDS, E = JUNGLE NOISES, F = FOOTSTEP

For the EFFECTS2 family, the high nibble is as follows:

1 = MACHINE GUN, 2 = TELEPHONE, 3 = DOG BARK, 4 = DOG GROWL, 5 = BOAT WHISTLE, 6 = OCEAN, 7 = WIND, 8 = CROWD BOOS, 9 = APPLAUSE, A = ROARING CROWDS, B = SCREAM, C = SWORD CLASH, D = AVALANCE, E = BOUNCING BALL, F = BALL AGAINST BAT OR CLUB

For the SYNTH family, the high nibble is as follows:

1 = STRINGS, 2 = SQUARE, 3 = SAWTOOTH, 4 = TRIANGLE, 5 = SINE, 6 = NOISE

So, for example if a wave's type byte was 0x26, this would be a SNARE DRUM. If a wave's type byte is 0, then this means "UNKNOWN" instrument.

==================== THE ORDER OF THE CHUNKS ========================

The SAMP header obviously must be first in the file, followed by the MHDR chunk. After this, the ANNO, (c), AUTH and NAME chunks may follow in any order, though none of these need appear in the file at all. The BODY chunk must be last.

================= FILENAME CONVENTIONS =================

For when it becomes necessary to split a SAMP file between floppies using the Continuation feature, the filenames should be related. The method is the following:

The "root" file has the name that the user chose to save under. Subsequent files have an ascii number appended to the name to indicate what sublevel the file is in. In this way, a program can reload the files in the proper order.

For example, if a user saved a file called "Gurgle", the first continuation file should be named "Gurgle1", etc.

============ WHY DOES ANYONE NEED SUCH A COMPLICATED FILE? ==============
(or "What's wrong with 8SVX anyway?")

In a nutshell, 8SVX is not adequate for professional music sampling. First of all, it is nearly impossible to use multi-sampling (utilizing several, different samples of any instrument throughout its musical range). This very reason alone makes it impossible to realistically reproduce a musical instrument, as none in existance (aside from an electronic organ) uses interpolations of a single wave to create its musical note range.

Also, stretching a sample out over an entire octave range does grotesque (and VERY unmusical) things to such elements as the overtone structure, wind/percussive noises, the instrument's amplitude envelope, etc. The 8SVX format is designed to stretch the playback in exactly this manner.

8SVX ignores MIDI which is the de facto standard of musical data transmission. 8SVX does not allow storing data for features that are commonplace to professional music samplers. Such features are: velocity sample start, separate filter and envelopes for each sample, separate sampling rates, and various playback modes like stereo sampling and panning.

SAMP attempts to remedy all of these problems with a format that can be used by a program that simulates these professional features in software. The format was inspired by the capabilities of the following musical products:

EMU's	EMAX, EMULATOR
SEQUENTIAL CIRCUIT's	PROPHET 2000, STUDIO 440
ENSONIQ's	MIRAGE
CASIO's	FZ-1
OBERHEIM's	DPX
YAMAHA	TX series

So why does the Amiga need the SAMP format? Because professional musician's are buying computers. With the firm establishment of MIDI, musician's are buying and using a variety of sequencers, patch editors, and scoring programs. It is now common knowledge amoung professional musicians that the Amiga lags far behind IBM clones, Macintosh, and Atari ST computers in both music software and hardware support. Both Commodore and the current crop of short-sighted 3rd party Amiga developers are pigeon-holing the Amiga as "a video computer". It is important for music software to exploit whatever capabilities the Amiga offers before the paint and animation programs, genlocks, frame-grabbers, and video breadboxes are the only applications selling for the Amiga. Hopefully, this format, with the SAMP disk I/O library will make it possible for Amiga software to attain the level of professionalism that the other machines now boast, and the Amiga lacks.

3-D rendering data, Turbo Silver (Impulse)

FORM TDDD

FORM TDDD is used by Impulse's Turbo Silver 3.0 for 3D rendering data. TDDD stands for "3D data description". The files contain object and (optionally) observer data.

Turbo Silver's successor, "Imagine", uses an upgraded FORM TDDD when it reads/writes object data.

Currently, in "standard IFF" terms, a FORM TDDD has only two chunk types: an INFO chunk describing observer data; and an OBJ chunk describing an object heirarchy. The INFO chunk appears only in Turbo Silver's "cell" files, and the OBJ chunk appears in both "cell" files and "object" files.

The FORM has an (optional) INFO chunk followed by some number of OBJ chunks. (Note: OBJ is followed by a space -- ckID = "OBJ ")

The INFO and OBJ chunks, in turn, are made up of smaller chunks with the standard IFF structure: <ID> <data-size> <data>.

The INFO "sub-chunks" are relatively straightforward to interpret.

The OBJ "sub-chunks" support object heirarchies, and are slightly more difficult to interpret. Currently, there are 3 types of OBJ sub-chunks: an EXTR chunk, describing an "external" object in a seperate file; a DESC chunk, describing one node of a heirarchy; and a TOBJ chunk marking the end of a heirarchy chain. For each DESC chunk, there must be a corresponding TOBJ chunk. And an EXTR chunk is equivalent to a DESC/TOBJ pair.

In Turbo Silver and Imagine, the structure of the object heirarchy is as follows. There is a head object, and its (sexist) brothers. Each brother may have child objects. The children may have grandchildren, and so on. The brother nodes are kept in a doubly linked list, and each node has a (possibly NULL) pointer to a doubly linked "child" list. The children point to the "grandchildren" lists, and so on. (In addition, each node has a "back" pointer to its parent).

Each of the "head" brothers is written in a seperate OBJ chunk, along with all its descendants. The descendant heirarchy is supported as follows:

```
    for each node of a doubly linked list,

    1)  A DESC chunk is written, describing its object.
    2)  If it has children, steps 1) to 3) are performed
            for each child.
    3)  A TOBJ chunk is written, marking the end of the children.
```

For "external" objects, steps 1) to 3) are not performed, but an EXTR chunk is written instead. (This means that an external object cannot have children unless they are stored in the same "external" file).

The TOBJ sub-chunks have zero size -- and no data. The DESC and EXTR sub-chunks are made up of "sub-sub-chunks", again, with the standard IFF structure: <ID> <data-size> <data>.

("External" objects were used by Turbo Silver to allow a its "cell" data files to refer to an "object" data file that is "external" to the cell file. Imagine abandons the idea of individual cell files, and deals only in TDDD "object" files. Currently, Imagine does not support EXTR chunks in TDD files.)

Reader software WILL FOLLOW the standard IFF procedure of skipping over any un-recognized chunks -- and "sub-chunks" or "sub-sub-chunks". The <data-size> field indicates how many bytes to skip. In addition it WILL OBSERVE the IFF rule that an odd <data-size> may appear, in which case the corredponding <data> field will be padded at the end with one extra byte to give it an even size.

Now, on with the details.

First, there are several numerical fields appearing in the data, describing object positions, rotation angles, scaling factors, etc. They are stored as "32-bit fractional" numbers, such that the true number is the 32-bit number divided by 65536. So as an example, the number 3.14159 is stored as (hexadecimal) $0003243F. This allows the data to be independant of any particular floating point format. And it (actually) is the internal format used in the "integer" version of Turbo Silver. Numbers stored in this format are called as "FRACT"s below.

Second, there are several color (or RGB) fields in the data. They are always stored as three UBYTEs representing the red, green and blue components of the color. Red is always first, followed by green, and then blue. For some of the data chunks, Turbo Silver reads the color field into the 24 LSB's of a LONGword. In such cases, the 3 RGB bytes are preceded by a zero byte in the file.

The following "typedef"s are used below:

```
typedef LONG    FRACT;                /* 4 bytes */
typedef UBYTE   COLOR[3];             /* 3 bytes */

typedef struct vectors {
    FRACT X;            /* 4 bytes */
    FRACT Y;            /* 4 bytes */
    FRACT Z;            /* 4 bytes */
} VECTOR;               /* 12 bytes total */

typedef struct matrices {
    VECTOR I;           /* 12 bytes */
    VECTOR J;           /* 12 bytes */
    VECTOR K;           /* 12 bytes */
} MATRIX;               /* 36 bytes total */

typedef struct _tform {
    VECTOR r;           /* 12 bytes - position */
    VECTOR a;           /* 12 bytes - x axis */
    VECTOR b;           /* 12 bytes - y axis */
    VECTOR c;           /* 12 bytes - z axis */
    VECTOR s;           /* 12 bytes - size */
} TFORM;                /*  60 bytes total */
```

The following structure is used in generating animated cells from a single cell. It can be attached to an object or to the camera. It is also used for Turbo Silver's "extrude along a path" feature. (It is ignored and forgotten by Imagine.)

```
typedef struct story {
    UBYTE  Path[18];  /* 18 bytes */
    VECTOR Translate; /* 12 bytes */
    VECTOR Rotate;    /* 12 bytes */
    VECTOR Scale;     /* 12 bytes */
    UWORD  info;      /*  2 bytes */
} STORY;              /* 56 bytes total */
```

The Path[] name refers to a named object in the cell data. The path object should be a sequence of points connected with edges. The object moves from the first point of the first edge, to the last point of the last edge. The edge ordering is important. The path is interpolated so that the object always moves an equal distance in each frame of the animation. If there is no path the Path[] field should be set to zeros.
The Translate vector is not currently used.
The Rotate "vector" specifies rotation angles about the X, Y, and Z axes.
The Scale vector specfies X,Y, and Z scale factors.
The "info" word is a bunch of bit flags:

```
    ABS_TRA    0x0001    - translate in world coorinates (not used)
    ABS_ROT    0x0002    - rotation in world coorinates
    ABS_SCL    0x0004    - scaling in world coorinates
    LOC_TRA    0x0010    - translate in local coorinates (not used)
    LOC_ROT    0x0020    - rotation in local coorinates
    LOC_SCL    0x0040    - scaling in local coorinates
    X_ALIGN    0x0100    - (not used)
    Y_ALIGN    0x0200    - align Y axis to path's direction
    Z_ALIGN    0x0400    - (not used)
    FOLLOW_ME  0x1000    - children follow parent on path
```

```
DESC sub-sub-chunks
-------------------

NAME - size 18

    BYTE    Name[18];       ; a name for the object.

    Used for camera tracking, specifying story paths, etc.

SHAP - size 4

    WORD    Shape;          ; number indicating object type
    WORD    Lamp;           ; number indicating lamp type

    Lamp numbers are composed of several bit fields:

            Bits 0-1:
        0 - not a lamp
        1 - like sunlight
        2 - like a lamp - intensity falls off with distance.
        3 - unused/reserved

            Bits 2:
                    0 - non-shadow-casting light
                    4 - shadow-casting light

            Bits 3-4:
        0  - Spherical light source
        8  - Cylindrical light source.
        16 - Conical light source.
        24 - unused/reserved
    Shape numbers are:

        0 - Sphere
        1 - Stencil          ; not supported by Imagine
        2 - Axis             ; custom objects with points/triangles
        3 - Facets           ; illegal - for internal use only
        4 - Surface          ; not supported by Imagine
        5 - Ground

    Spheres have thier radius set by the X size parameter.
    Stencils and surfaces are plane-parallelograms, with one
    point at the object's position vector; one side lying along
    the object's X axis with a length set by the X size; and
    another side starting from the position vector and going
    "Y size" units in the Y direction and "Z size" units in
    the X direction.  A ground object is an infinte plane
    perpendicular to the world Z axis.  Its Z coordinate sets
    its height, and the X and Y coordinates are only relevant
    to the position of the "hot point" used in selecting the
    object in the editor.  Custom objects have points, edges
    and triangles associated with them.  The size fields are
    relevant only for drawing the object axes in the editor.
    Shape number 3 is used internally for triangles of custom
    objects, and should never appear in a data file.

POSI - size 12

    VECTOR  Position;       ; the object's position.

    Legal coordinates are in the range -32768 to 32767 and 65535/65536.
    Currently, the ray-tracer only sees objects in the -1024 to 1024
    range.  Light sources, and the camera may be placed outside that
    range, however.

AXIS - size 36

    VECTOR  XAxis;
    VECTOR  YAxis;
    VECTOR  ZAxis;

    These are direction vectors for the object coordinate system.
    They must be "orthogonal unit vectors" - i.e. the sum of the
    squares of the vevtor components must equal one (or close to it),
    and the vectors must be perpendicular.

SIZE - size 12

    VECTOR  Size;

    See SHAP chunk above.  The sizes are used in a variety of ways
    depending on the object shape.  For custom objects, they are
    the lengths of the coordinate axes drawn in the editor.  If the
    object has its "Quickdraw" flag set, the axes lengths are also
    used to set the size of a rectangular solid that is drawn rather
    than drawing all the points and edges.

PNTS - size 2 + 12 * point count

    UWORD   PCount;         ; point count
    VECTOR  Points[];       ; points

    This chunk has all the points for custom objects.  The are
    refered to by thier position in the array.
```

```
EDGE - size 4 + 4 * edge cout

    UWORD   ECount;         ; edge count
    UWORD   Edges[][2];     ; edges

    This chunk contins the edge list for custom objects.
    The Edges[][2] array is pairs of point numbers that
    are connected by the edges.  Edges are refered to by thier
    position in the Edges[] array.

FACE - size 2 + 6 * face count

    UWORD   TCount;         ; face count
    UWORD   Connects[][3];  ; faces

    This chunk contains the triangle (face) list for custom objects.
    The Connects[][3] array is triples of edge numbers that are
    connected by triangles.

PTHD - size 2 + 6 * axis count - Imagine only

    UWORD   ACount;         ; axis count
    TFORM   PData[][3];     ; axis data

    This chunk contains the axis data for Imagine "path" objects.
    The PData array contains a TFORM structure for each point along
    the path.  The "Y size" item for the last point on the path tells
    whether the path is closed or not.  Zero means closed, non-zero
    means open.  Otherwise the Y size field is the distance along
    the path to the next path point/axis.

COLR - size 4
REFL - size 4
TRAN - size 4
SPC1 - size 4 - Imagine only

    BYTE    pad;            ; pad byte - must be zero
    COLOR   col;            ; RGB color

    These are the main object RGB color, and reflection, transmission
    and specularity coefficients.

CLST - size 2 + 3 * count
RLST - size 2 + 3 * count
TLST - size 2 + 3 * count

    UWORD   count;          ; count of colors
    COLOR   colors[];       ; colors

    These are the color, reflection and transmission coefficients
    for each face in custom objects. The count should match the
    face count in the FACE chunk. The ordering corresponds to the
    face order.

TPAR - size 64 - not written by Imagine - see TXT1 below

    FRACT   Params[16];     ; texture parameters

    This is the list of parameters for texture modules when
    texture mapping is used.
```

```
TXT1 - variable size - Imagine only

    This chunk contains texture data when texture mapping is used.

    UWORD   Flags;           ; texture flags:
                             ;    1 - TXTR_CHILDREN - apply to child objs
    TFORM   TForm;           ; local coordinates of texture axes.
    FRACT   Params[16];      ; texture parameters
    UBYTE   PFlags[16];      ; parameter flags (currently unused)
    UBYTE   Length;          ; length of texture file name
    UBYTE   Name[Length];    ; texture file name (not NULL terminated)
    UBYTE   pad;             ; (if necessary to make an even length)

BRS1 - variable size - Imagine only (version 1.0)
BRS2 - variable size - Imagine only (version 1.1)

    UWORD   Flags;           ; brush type:
                             ;    0 - Color
                             ;    1 - Reflection
                             ;    2 - Filter
                             ;    3 - Altitude
    UWORD   WFlags;          ; brush wrapping flags:
                             ;    1  WRAP_X        - wrap type
                             ;    2  WRAP_Z        - wrap type
                             ;    4  WRAP_CHILDREN - apply to children
                             ;    8  WRAP_REPEAT   - repeating brush
                             ;    16 WRAP_FLIP     - flip with repeats
    TFORM   TForm;           ; local coordinates of brush axes.
    (UWORD   FullScale;)     ; full scale value
    (UWORD   MaxSeq;)        ; highest number for sequenced brushes
    UBYTE   Length;          ; length of brush file name
    UBYTE   Name[Length];    ; brush file name (not NULL terminated)
    UBYTE   pad;             ; (if necessary to make an even length)

    The FullScale and MaxSeq items are in BRS2 chunks only.

SURF - size 5 - not written by Imagine

    BYTE    SProps[5];       ; object properties

    This chunk contains object (surface) properties used by Turbo Silver.

    SProps[0] - PRP_SURFACE ; surface type
                            ;   0 - normal
                            ;   4 - genlock
                            ;   5 - IFF brush
    SProps[1] - PRP_BRUSH   ; brush number (if IFF mapped)
    SProps[2] - PRP_WRAP    ; IFF brush wrapping type
                            ;   0 - no wrapping
                            ;   1 - wrap X
                            ;   2 - wrap Z
                            ;   3 - wrap X and Z
    SProps[3] - PRP_STENCIL ; stencil number for stencil objects
    SProps[4] - PRP_TEXTURE ; texture number if texture mapped

MTTR - size 2 - not written by Imagine - see PRP1 chunk.

    UBYTE   Type;            ; refraction type (0-4)
    UBYTE   Index;           ; custom index of refraction

    This chunk contains refraction data for transparent or glossy
    objects.  If the refraction type is 4, the object has a "custom"
    refractive index stored in the Index field.  The Index field is
    100 * (true index of refraction - 1.00) -- so it must be in the
    range of 1.00 to 3.55.  The refraction types 0-3 specify 0) Air
    - 1.00, 1) Water - 1.33, 2) Glass - 1.67 or 3) Crystal 2.00.
```

```
SPEC - size 2 - not written by Imagine - see SPC1 above.

    UBYTE   Specularity;     ; range of 0-255
    UBYTE   Hardness;        ; specular exponent (0-31)

    This chunk contains specular information.  The Specularity
    field is the amount of specular reflection -- 0 is none,
    255 is fully specular.  The "specular exponent" controls
    the "tightness" of the specular spots.  A value of zero
    gives broad specular spots and a value of 31 gives smaller
    spots.

PRP0 - size 6 - not written by Imagine

    UBYTE   Props[6];        ; more object properties

    This chunk contains object properties that programs other
    than Turbo Silver might support.

    Props[0] - PRP_BLEND    ; blending factor (0-255)
    Props[1] - PRP_SMOOTH   ; roughness factor
    Props[2] - PRP_SHADE    ; shading on/off flag
    Props[3] - PRP_PHONG    ; phong shading on/off flag
    Props[4] - PRP_GLOSSY   ; glossy on/off flag
    Props[5] - PRP_QUICK    ; Quickdraw on/off flag

    The blending factor controls the amount of dithering used
    on the object - 255 is fully dithered.  The roughness factor
    controls how rough the object should appear - 0 is smooth, 255
    is max roughness.  The shading flag is interpreted differently
    depending on whether the object is a light source or not.  For
    light sources, it sets the light to cast shadows or not.  For
    normal objects, if the flag is set, the object is always
    considered as fully lit - i.e., it's color is read directly
    from the object (or IFF brush), and is not affected by light
    sources.  The phong shading is on by default - a non-zero value
    turns it off.  The glossy flag sets the object to be glossy or
    not.  If the object is glossy, the "transmit" colors and the
    index of refraction control the amount of "sheen".  The glossy
    feature is meant to simulate something like a wax coating
    on the object with the specified index of refraction. The
    trasmission coefficients control how much light from the
    object makes it through the wax coating.
    The Quickdraw flag, if set, tells the editor not to draw
    all the points and edges for the object, but to draw a
    rectanglular solid centered at the object position, and
    with sizes detemined by the axis lengths.

PRP1 - size 8 - Imagine only

    UBYTE   IProps[8];       ; more object properties

    This chunk contains object properties that programs other
    than Imagine might support.

    IProps[0] - IPRP_DITHER   ; blending factor (0-255)
    IProps[1] - IPRP_HARD     ; hardness factor (0-255)
    IProps[2] - IPRP_ROUGH    ; roughness factor (0-255)
    IProps[3] - IPRP_SHINY    ; shinyness factor (0-255)
    IProps[4] - IPRP_INDEX    ; index of refraction
    IProps[5] - IPRP_QUICK    ; flag - Quickdraw on/off
    IProps[6] - IPRP_PHONG    ; flag - Phong shading on/off
    IProps[7] - IPRP_GENLOCK  ; flag - Genlock on/off
    The blending factor controls the amount of dithering used
    on the object - 255 is fully dithered.
    The hardness factor controls how tight the specular spot
    should be - 0 is a big soft spot, 255 is a tight hot spot
    The roughness factor controls how rough the object should
    appear - 0 is smooth, 255 is max roughness.
    The shiny factor in interaction with the object's filter
    values controls how shiny the object appears.  Setting it
    to anything but zero forces the object to be non-transparent
    since then the filter values are used in the shiny (reflection)
    calculations.  A value of 255 means maximum shinyness.

INTS - size 4 - not written by Imagine

    FRACT   Intensity;      ; light intensity

    This is the intensity field for light source objects.
    an intensity of 255 for a sun-like light fully lights
    object surfaces which are perpendicular to the direction
    to the light source.  For lamp-like light sources, the
    necessary intensity will depend on the distance to the light.

INT1 - size 12 - Imagine only

    VECTOR  Intensity;      ; light intensity

    This is like INTS above, but has seperate R, G & B intensities.

STRY - size 56 - not written by Imagine

    STORY   story;          ; a story structure for the object.

    The story structure is described above.

ANID - size 64 - Imagine only

    LONG    Cellno;         ; cell number
    TFORM   TForm;          ; object position/axes/size in that cell.

    For Imagine's "Cycle" objects, within EACH DESC chunk in the
    file - that is, for each object of the group, there will be
    a series of ANID chunks.  The cell number sequences of each
    part of the must agree with the sequence for the head object,
    and the first cell number must be zero.

FORD - size 56 + 12 * PC - Imagine only

    WORD    NumC;           ; number of cross section points
    WORD    NumF;           ; number of slices
    WORD    Flags;          ; orientation flag
    WORD    pad;            ; reserved
    MATRIX  TForm;          ; object rotation/scaling transformation
    VECTOR  Shift;          ; object translation
    VECTOR  Points[PC];     ; "Forms" editor points

    For Imagine's "Forms" objects, the "PNTS" chunk above is not
    written out, but this structure is written instead.  The point
    count is PC = NumC + 4 * NumF.  The object's real points are
    then calculated from these using a proprietary algorithm.
    The tranformation parameters above allow the axes of the
    real object be moved around relative to the "Forms" points.
```

```
DESC notes
----------

Again, most of these fields are optional, and defaults are supplied.
However, if there is a FACE chunk, there must also be a CLST chunk,
an RLST chunk and a TLST chunk -- all with matching "count" fields.
The SHAP chunk is not optional.

Defaults are:  Colors set to (240,240,240); reflection and
transmission coefficients set to zero; illegal shape; no story or
special surface types; position at (0,0,0); axes aligned to the
world axes; size fields all 32.0; intensity at 300; no name;
no points/edges or faces; texture parameters set to zero; refraction
type 0 with index 1.00; specular, hardness and roughness set to zero;
blending at 255; glossy off; phong shading on; not a light source;
not brightly lit;

In Imagine, defaults are the same, but with colors (255,255,255).


INFO sub-chunks
---------------

BRSH - size 82

    WORD    Number;         ; Brush number (between 0 and 7)
    CHAR    Filename[80];   ; IFF ILBM filename

    There may be more than one of these.

STNC - size 82

    Same format as BRSH chunk.

TXTR - size 82

    Same format as BRSH chunk.  The Filename field is the name of
    a code module that can be loaded with LoadSeg().

OBSV - size 28

    VECTOR  Camera;         ; Camera position
    VECTOR  Rotate;         ; Camera rotation angles
    FRACT   Focal;          ; Camera focal length

    This tells where the camera is, how it is aimed, and its
    focal length.  The rotation angles are in degrees, and specify
    rotations around the X, Y, and Z axes.  The camera looks down
    its own Y axis, with the top of the picture in the direction of
    the Z axis.  If the rotation angles are all zero, its axes
    are aligned with the world coordinate axes.  The rotations are
    performed in the order ZXY about the camera axes.  A positive
    angle rotates Y toward Z, Z toward X, and X toward Y for
    rotations about the X, Y, and Z axes respectively.  To
    understand the focal length, imagine a 320 x 200 pixel
    rectangle perpendicular to, and centered on the camera's
    Y axis.  Any objects in the infinite rectangular cone defined
    by the camera position and the 4 corners of the rectangle will
    appear in the picture.
```

```
OTRK - size 18

    BYTE    Trackname[18];

    This chunk specifies the name of an object that the camera is
    "tracked" to.  If the name is NULL, the camera doesn't track
    Otherwise, if the object is moved inside Turbo Silver, the
    camera will follow it.

OSTR - size 56

    STORY   CStory;          ; a STORY structure for the camera

    The story structure is defined above.

FADE - size 12

    FRACT   FadeAt;          ; distance to start fade
    FRACT   FadeBy;          ; distance of total fade
    BYTE    pad;             ; pad byte - must be zero
    COLOR   FadeTo;          ; RGB color to fade to

SKYC - size 8

    BYTE    pad;             ; pad byte - must be zero
    COLOR   Horizon;         ; horizon color
    BYTE    pad;             ; pad byte - must be zero
    COLOR   Zenith;          ; zenith color

AMBI - size 4

    BYTE    pad;             ; pad byte - must be zero
    COLOR   Ambient;         ; abmient light color

GLB0 - size 8

    BYTE    Props[8];        ; an array of 8 "global properties" used
                             ; by Turbo Silver.

    Props[0] - GLB_EDGING        ; edge level (globals requester)
    Props[1] - GLB_PERTURB       ; perturbance (globals requester)
    Props[2] - GLB_SKY_BLEND     ; sky blending factor (0-255)
    Props[3] - GLB_LENS          ; lens type (see below)
    Props[4] - GLB_FADE          ; flag - Sharp/Fuzzy focus (globals)
    Props[5] - GLB_SIZE          ; "apparant size" (see below)
    Props[6] - GLB_RESOLVE       ; resolve depth (globals requester)
    Props[7] - GLB_EXTRA         ; flag - genlock sky on/off

    The edging and perturbance values control the heuristics in ray
    tracing.  The sky blending factor is zero for no blending, and 255
    for full blending.  The lens type is a number from 0-4, corresponding
    to the boxes in the "camera" requester, and correspond to 0) Manual,
    1) Wide angle, 2) Normal, 3) Telephoto, and 4) Custom.  It is used
    in setting the camera's focal length if the camera is tracked to an
    object.  The Sharp/Fuzzy flag turns the "fade" feature on and off -
    non-zero means on.  The "apparant size" parameter is 100 times the
    "custom size" parameter in the camera requester.  And is used to set
    the focal length for a custom lens.  The "resolve depth" controls
    the number of rays the ray tracer will shoot for a single pixel.
    Each reflective/refractive ray increments the depth counter, and
    the count is never allowed to reach the "resolve depth".  If both
    a reflective and a refractive ray are traced, each ray gets its
    own version of the count - so theoretically, a resolve depth of
    4 could allow much more than 4 rays to be traced.  The "genlock
    sky" flag controls whether the sky will be colored, or set to
    the genlock color (color 0 - black) in the final picture.
```

```
All of the INFO sub-chunks are optional, as is the INFO chunk.
Default values are supplied if the chunks are not present.  The
defaults are:  no brushes, stencils, or textures defined; no story
for the camera; horizon and zenith and ambient light colors set
to black; fade color set to (80,80,80);  un-rotated, un-tracked
camera at (-100, -100, 100); and global properties array set to
[30, 0, 0, 0, 0, 100, 8, 0].


EXTR sub-sub-chunks
-------------------

MTRX - size 60

    VECTOR  Translate;       ; translation vector
    VECTOR  Scale;           ; X,Y and Z scaling factors
    MATRIX  Rotate;          ; rotation matrix

    The translation vector is i world coordinates.
    The scaling factors are with respect to local axes.
    The rotation matrix is with respect to the world axes,
    and it should be a "unit matrix".
    The rotation is such that a rotated axis's X,Y, and Z
    components are the dot products of the MATRIX's I,J,
    and K vectors with the un-rotated axis vector.

LOAD - size 80

    BYTE    Filename[80];    ; the name of the external file

    This chunk contains the name of an external object file.
    The external file should be a FORM TDDD file.  It may contain
    an any number of objects possibly grouped into heirarchy(ies).

Both of these chunks are required.
```

WORD.doc Page 1

```
ProWrite document format (New Horizons)

TITLE:  WORD  (word processing FORM used by ProWrite)

IFF FORM / CHUNK DESCRIPTION
============================

Form/Chunk IDs:
   FORM   WORD
   Chunks FONT,COLR,DOC,HEAD,FOOT,PCTS,PARA,TABS,PAGE,TEXT,FSCC,PINF

Date Submitted: 03/87
Submitted by:   James Bayless - New Horizons Software, Inc.


FORM
====

FORM ID:  WORD

FORM Purpose:  Document storage (supports color, fonts, pictures)

FORM Description:

This include file describes FORM WORD and its Chunks

/*
 *      IFF Form WORD structures and defines
 *      Copyright (c) 1987 New Horizons Software, Inc.
 *
 *      Permission is hereby granted to use this file in any and all
 *      applications.  Modifying the structures or defines included
 *      in this file is not permitted without written consent of
 *      New Horizons Software, Inc.
 */

#include ":IFF/ILBM.h"        /* Makes use of ILBM defines */

#define ID_WORD      MakeID('W','O','R','D')      /* Form type */

#define ID_FONT      MakeID('F','O','N','T')      /* Chunks */
#define ID_COLR      MakeID('C','O','L','R')
#define ID_DOC       MakeID('D','O','C',' ')
#define ID_HEAD      MakeID('H','E','A','D')
#define ID_FOOT      MakeID('F','O','O','T')
#define ID_PCTS      MakeID('P','C','T','S')
#define ID_PARA      MakeID('P','A','R','A')
#define ID_TABS      MakeID('T','A','B','S')
#define ID_PAGE      MakeID('P','A','G','E')
#define ID_TEXT      MakeID('T','E','X','T')
#define ID_FSCC      MakeID('F','S','C','C')
#define ID_PINF      MakeID('P','I','N','F')

/*
 *   Special text characters for page number, date, and time
 *   Note:  ProWrite currently supports only PAGENUM_CHAR, and only in
 *      headers and footers
 */

#define PAGENUM_CHAR   0x80
#define DATE_CHAR      0x81
#define TIME_CHAR      0x82
```

WORD.doc Page 2

```
/*
 *   Chunk structures follow
 */

/*
 *   FONT - Font name/number table
 *   There are one of these for each font/size combination
 *   These chunks should appear at the top of the file (before document data)
 */

typedef struct {
   UBYTE   Num;          /* 0 .. 255 */
   UWORD   Size;
/* UBYTE   Name[];       */   /* NULL terminated, without ".font" */
} FontID;

/*
 *   COLR - Color translation table
 *   Translates from color numbers used in file to ISO color numbers
 *   Should be at top of file (before document data)
 *   Note:  Currently ProWrite only checks these values to be its current map,
 *      it does no translation as it does for FONT chunks
 */

typedef struct {
   UBYTE   ISOColors[8];
} ISOColors;

/*
 *   DOC - Begin document section
 *   All text and paragraph formatting following this chunk and up to a
 *   HEAD, FOOT, or PICT chunk belong to the document section
 */

#define PAGESTYLE_1   0      /* 1, 2, 3 */
#define PAGESTYLE_I   1      /* I, II, III */
#define PAGESTYLE_i   2      /* i, ii, iii */
#define PAGESTYLE_A   3      /* A, B, C */
#define PAGESTYLE_a   4      /* a, b, c */

typedef struct {
   UWORD   StartPage;      /* Starting page number */
   UBYTE   PageNumStyle;   /* From defines above */
   UBYTE   pad1;
   LONG    pad2;
} DocHdr;

/*
 *   HEAD/FOOT - Begin header/footer section
 *   All text and paragraph formatting following this chunk and up to a
 *   DOC, HEAD, FOOT, or PICT chunk belong to this header/footer
 *   Note:  This format supports multiple headers and footers, but currently
 *      ProWrite only allows a single header and footer per document
 */

#define PAGES_NONE   0
#define PAGES_LEFT   1
#define PAGES_RIGHT  2
#define PAGES_BOTH   3

typedef struct {
   UBYTE   PageType;       /* From defines above */
   UBYTE   FirstPage;      /* 0 = Not on first page */
   LONG   pad;
} HeadHdr;
```

```
/*
 *   PCTS - Begin picture section
 *   Note:  ProWrite currently requires NPlanes to be three (3)
 */

typedef struct {
   UBYTE   NPlanes;      /* Number of planes used in picture bitmaps */
   UBYTE   pad;
} PictHdr;

/*
 *   PARA - New paragraph format
 *   This chunk should be inserted first when a new section is started (DOC,
 *      HEAD, or FOOT), and again whenever the paragraph format changes
 */

#define SPACE_SINGLE   0
#define SPACE_DOUBLE   0x10

#define JUSTIFY_LEFT    0
#define JUSTIFY_CENTER  1
#define JUSTIFY_RIGHT   2
#define JUSTIFY_FULL    3

#define MISCSTYLE_NONE   0
#define MISCSTYLE_SUPER  1      /* Superscript */
#define MISCSTYLE_SUB    2      /* Subscript */

typedef struct {
   UWORD   LeftIndent;    /* In decipoints (720 dpi) */
   UWORD   LeftMargin;
   UWORD   RightMargin;
   UBYTE   Spacing;       /* From defines above */
   UBYTE   Justify;       /* From defines above */
   UBYTE   FontNum;       /* FontNum, Style, etc. for first char in para*/
   UBYTE   Style;         /* Standard Amiga style bits */
   UBYTE   MiscStyle;     /* From defines above */
   UBYTE   Color;         /* Internal number, use COLR to translate */
   LONG    pad;
} ParaFormat;

/*
 *   TABS - New tab stop types/locations
 *   Use an array of values in each chunk
 *   Like the PARA chunk, this should be inserted whenever the tab settings
 *      for a paragraph change
 *   Note:  ProWrite currently does not support TAB_CENTER
 */

#define TAB_LEFT     0
#define TAB_CENTER   1
#define TAB_RIGHT    2
#define TAB_DECIMAL  3

typedef struct {
   UWORD   Position;      /* In decipoints */
   UBYTE   Type;
   UBYTE   pad;
} TabStop;

/*
 *   PAGE - Page break
 *   Just a marker -- this chunk has no data
 */
```

```
/*
 *   TEXT - Paragraph text (one block per paragraph)
 *   Block is actual text, no need for separate structure
 *   If the paragraph is empty, this is an empty chunk -- there MUST be
 *   a TEXT block for every paragraph
 *   Note:  The only ctrl characters ProWrite can currently handle in TEXT
 *   chunks are Tab and PAGENUM_CHAR, ie no Return's, etc.
 */

/*
 *   FSCC - Font/Style/Color changes in previous TEXT block
 *   Use an array of values in each chunk
 *   Only include this chunk if the previous TEXT block did not have
 *      the same Font/Style/Color for all its characters
 */

typedef struct {
   UWORD   Location;      /* Character location in TEXT chunk of change */
   UBYTE   FontNum;
   UBYTE   Style;
   UBYTE   MiscStyle;
   UBYTE   Color;
   UWORD   pad;
} FSCChange;

/*
 *   PINF - Picture info
 *   This chunk must only be in a PCTS section
 *   Must be followed by ILBM BODY chunk
 *   Pictures are treated independently of the document text (like a
 *      page-layout system), this chunk includes information about what
 *      page and location on the page the picture is at
 *   Note:  ProWrite currently only supports mskTransparentColor and
 *      mskHasMask masking
 */

typedef struct {
   UWORD         Width, Height;   /* In pixels */
   UWORD         Page;         /* Which page picture is on (0..max) */
   UWORD         XPos, YPos;      /* Location on page in decipoints */
   Masking       Masking;      /* Like ILBM format */
   Compression   Compression;   /* Like ILBM format */
   UBYTE         TransparentColor;   /* Like ILBM format */
   UBYTE         pad;
} PictInfo;

/* end */
```

```
          Intro to IFF Amiga ILBM Files and Amiga Viewmodes
          =================================================

The IFF (Interchange File Format) for graphic images on the Amiga is called
FORM ILBM (InterLeaved BitMap).  It follows a standard parsable IFF format.

Sample hex dump of beginning of an ILBM:
========================================

Important note!  You can NOT ever depend on any particular ILBM chunk being
at any particular offset into the file!  IFF files are composed, in their
simplest form, of chunks within a FORM.  Each chunk starts starts with a
4-letter chunkID, followed by a 32-bit length of the rest of the chunk.  You
PARSE IFF files, skipping past unneeded or unknown chunks by seeking their
length (+1 if odd length) to the next 4-letter chunkID.

0000: 464F524D 00016418 494C424D 424D4844    FORM..d.ILBMBMHD
0010: 00000014 01400190 00000000 06000100    .....@..........
0020: 00000A0B 01400190 43414D47 00000004    .....@..CAMG....
0030: 00000804 434D4150 00000030 001100EE    ....CMAP...0....
0040: EEEE0000 22000055 33333355 55550033    .... ..P000PPP.0
0050: 99885544 77777711 66EE2266 EE6688DD    ..P@ppp.`. `.`..
0060: AAAAAAAA 99EECCCC CCDDAAEE 424F4459    ............BODY
0070: 000163AC F8000F80 148A5544 2ABDEFFF    ..c.......UD*...  etc.

Interpretation:

      'F O R M' length  'I L B M''B M H D'<-start of BitMapHeader chunk
0000: 464F524D 00016418 494C424D 424D4844    FORM..d.ILBMBMHD

       length  WideHigh XorgYorg PlMkCoPd <- Planes Mask Compression Pad
0010: 00000014 01400190 00000000 06000100    .....@..........

      TranAspt PagwPagh 'C A M G' length  <- start of C-AMiGa View modes chunk
0020: 00000A0B 01400190 43414D47 00000004    .....@..CAMG....

      Viewmode 'C M A P' length  R g b R  <- Viewmode 800=HAM | 4=LACE
0030: 00000804 434D4150 00000030 001100EE    ....CMAP...0....

      g b R g  b R g b  R g b R  g b R g  <- Rgb's are for reg0 thru regN
0040: EEEE0000 22000055 33333355 55550033    .... ..P000PPP.0

      b R g b  R g b R  g b R g  b R g b
0050: 99885544 77777711 66EE2266 EE6688DD    ..P@ppp.`. `.`..

      R g b R  g b R g  b R g b 'B O D Y'
0060: AAAAAAAA 99EECCCC CCDDAAEE 424F4459    ............BODY

       length   start of body data        <- Compacted (Compression=1 above)
0070: 000163AC F8000F80 148A5544 2ABDEFFF    ..c.......UD*...
0080: FFBFF800 0F7FF7FC FF04F85A 77AD5DFE    ...........Zw.].  etc.

Notes on CAMG Viewmodes:  HIRES=0x8000  LACE=0x4  HAM=0x800  HALFBRITE=0x80
```

```
Interpreting ILBMs
==================

ILBM is a fairly simple IFF FORM.  All you really need to deal with to
extract the image are the following chunks:

(Note - Also watch for AUTH Author chunks and (c) Copyright chunks
 and preserve any copyright information if you rewrite the ILBM)

   BMHD - info about the size, depth, compaction method
          (See interpreted hex dump above)

   CAMG - optional Amiga viewmodes chunk
          Most HAM and HALFBRITE ILBMs should have this chunk.  If no
          CAMG chunk is present, and image is 6 planes deep, assume
          HAM and you'll probably be right.  Some Amiga viewmodes
          flags are HIRES=0x8000, LACE=0x4, HAM=0x800, HALFBRITE=0x80.
          Note that new Amiga 2.0 ILBMs may have more complex 32-bit
          numbers (modeid) stored in the CAMG.  However, the bits
          described above should get you a compatible old viewmode.

   CMAP - RGB values for color registers 0 to n
          (each component left justified in a byte)
          If a deep ILBM (like 12 or 24 planes), there should be no CMAP
          and instead the BODY planes are interpreted as the bits of RGB
          in the order R0...Rn G0...Gn B0...Bn

   BODY - The pixel data, stored in an interleaved fashion as follows:
          (each line individually compacted if BMHD Compression = 1)
             plane 0 scan line 0
             plane 1 scan line 0
             plane 2 scan line 0
             ...
             plane n scan line 0
             plane 0 scan line 1
             plane 1 scan line 1
             etc.


Body Compression
================

The BODY contains pixel data for the image.  Width, Height, and depth
(Planes) is specified in the BMHD.

If the BMHD Compression byte is 0, then the scan line data is not compressed.
If Compression=1, then each scan line is individually compressed as follows:

   More than 2 bytes the same stored as BYTE code value n from  -1 to -127
     followed by byte to be repeated (-n) + 1 times.
   Varied bytes stored as BYTE code n from 0 to 127 followed by n+1 bytes
     of data.
   The byte code -128 is a NOP.
```

```
Interpreting the Scan Line Data:
================================

If the ILBM is not HAM or HALFBRITE, then after parsing and uncompacting if
necessary, you will have N planes of pixel data.  Color register used for
each pixel is specified by looking at each pixel thru the planes.  I.e.,
if you have 5 planes, and the bit for a particular pixel is set in planes
0 and 3:

       PLANE     4 3 2 1 0
       PIXEL     0 1 0 0 1

then that pixel uses color register binary 01001 = 9

The RGB value for each color register is stored in the CMAP chunk of the
ILBM, starting with register 0, with each register's RGB value stored as
one byte of R, one byte G, and one byte of B, with each component scaled
to 8-bits.  (ie. 4-bit Amiga R, G, and B components are each stored in the
high nibble of a byte.  The low nibble may also contain valid data if the
color was stored with 8-bit-per-gun color resolution).


BUT - if the picture is HAM or HALFBRITE, it is interpreted differently.
                        ===    =========

Hopefully, if the picture is HAM or HALFBRITE, the package that saved it
properly saved a CAMG chunk (look at a hex dump of your file with ACSII
interpretation - you will see the chunks - they all start with a 4-ASCII-
character chunk ID).  If the picture is 6 planes deep and has no CAMG chunk,
it is probably HAM.   If you see a CAMG chunk, the "CAMG" is followed by the
32-bit chunk length, and then the 32-bit Amiga Viewmode flags.

HAM pics with a 16-bit CAMG will have the 0x800 bit set in CAMG ViewModes.
HALBRITE pics will have the 0x80 bit set.

To transport a HAM or HALFBRITE picture to another machine, you must
understand how HAM and HALFBRITE work on the Amiga.


How Amiga HAM mode works:
=========================

Amiga HAM (Hold and Modify) mode lets the Amiga display all 4096 RGB values.
In HAM mode, the bits in the two last planes describe an R G or B
modification to the color of the previous pixel on the line to create the
color of the current pixel.  So a 6-plane HAM picture has 4 planes for
specifying absolute color pixels giving up to 16 absolute colors which would
be specified in the ILBM CMAP chunk.  The bits in the last two planes are
color modification bits which cause the Amiga, in HAM mode, to take the RGB
value of the previous pixel (Hold and), substitute the 4 bits in planes 0-3
for the previous color's R G or B component (Modify) and display the result
for the current pixel.  If the first pixel of a scan line is a modification
pixel, it modifies the RGB value of the border color (register 0).  The color
modification bits in the last two planes (planes 4 and 5) are interpreted as
follows:

   00 - no modification.  Use planes 0-3 as normal color register index
   10 - hold previous, replacing Blue component with bits from planes 0-3
   01 - hold previous, replacing Red component with bits from planes 0-3
   11 - hold previous. replacing Green component with bits from planes 0-3
```

```
How Amiga HALFBRITE mode works:
===============================

This one is simpler.  In HALFBRITE mode, the Amiga interprets the bit in the
last plane as HALFBRITE modification.  The bits in the other planes are
treated as normal color register numbers (RGB values for each color register
is specified in the CMAP chunk).  If the bit in the last plane is set (1),
then that pixel is displayed at half brightness.  This can provide up to 64
absolute colors.


Other Notes:
============

Amiga ILBMs images must be a even number of bytes wide.  Smaller images (such
as brushes) are padded to an even byte width.

ILBMs created with Electronic Arts IBM and Amiga "DPaintII" packages are
compatible (though you may have to use a '.lbm' filename extension on an
IBM).  The ILBM graphic files may be transferred between the machines (or
between the Amiga and IBM sides your Amiga if you have a CBM Bridgeboard
card installed) and loaded into either package.
```

IFF Source Code

This section contains a variety of source code listings showing how to use IFF files in applications. All of these programs require the new *iffparse.library* included with Release 2.0 of the Amiga operating system (Kickstart V36 and greater). There are four parts:

- IFF include files. These have been updated to be compatible with iffparse.library.
- Link modules which provide convenient IFF handling routines such as *showilbm.c*.
- Example programs showing how to use the link modules.
- Stand-alone utility and example programs.

IFFP_Modules.README Page 1

```
                    IFFP Modules - July 1991
                         version 37.5


These iffparse.library code modules and examples are designed as
replacements for the original EA IFF code.  In some modules, it has
been possible to retain much of the original code.  However, most
structures and most higher level function interfaces have changed.


On the plus side, these new modules contain many new high-level
easy-to-use functions for querying, loading, displaying, and saving
ILBMs.  During their development, modules similar to these have been
used inhouse at Commodore for the 2.0 Display program and several other
ILBM applications.  The screen.c module provides powerful display-opening
functions which are 1.3-compatible yet provide a host of new options under
2.0 such as centered overscan screens, full-video display clips, border
transparency control, and autoscroll.  New modules have been added for
printing (screendump) and for preserving/adding chunks (copychunks).
And the 8SVX example now actually plays samples and instruments.
In addition, clipboard support is automatic for all applications that
use the IFFP modules because parse.c's openifile() interprets the
filename -c[n] (ie. "-c", "-c1", "-c2", etc.) as clipboard unit n.


All of the applications presented here require iffparse.library which
is distributed on Workbench 2.0.  Please note that iffparse.library is
a 1.3-compatible library, and that all of these modules and examples
have been designed to take advantage of 2.0, but also work under 1.3.
Developers who wish to distribute iffparse.library on their commercial
products may execute a 2.0 Workbench license, or may get an addendum to
their 1.3 Workbench license to allow distribution of iffparse.library.


It was not necessary to port the gio IO speedup code since iffparse
can use your compiler's own buffered IO routines through the callback
stdio_stream() in parse.c.  Depending on your application, you may want
to add your own additional buffering to this stdio_stream() code.


Most of the high-level function pairs provided in these modules have
been designed to provide safe cleanup for themselves.  For example,
a loadbrush() that succeeds or fails at any point can be cleaned up
via unloadbrush.  The cleanup routines null out the appropriate
pointers so that allocations will not be freed twice.


All applications are built upon the parse.c module.  The basic concept
of using the parse.c module are:

        - Define tag-like arrays of your desired chunks (readers only)
        - Allocates one or more [form]Info structures as defined in
           iffp/[form]app.h (for example an ILBMInfo defined in
           iffp/ilbmapp.h).
        - Initialize the ParseInfo part of these structures to the desired
           chunk arrays, and to an IFFHandle allocated via iffparse
           AllocIFF().
        - Use the provided high level load/save functions, or use the
           lower level parse.c openifile(), reader-only parseifile()/
           getcontext()/nextcontext(), and closeifile().  The filename
           -c[n] may be used to read/write clipboard unit n.
        - Clean up, FreeIFF(), and deallocate [form]Info's.
```

IFFP_Modules.README Page 2

```
IMPORTANT NOTES - Most of the higher-level load functions keep the
        IFFHandle (file or clipboard) open.  While the handle is
        open, you may use parse.c functions (such as findpropdata)
        OR direct iffparse functions (FindProp(), FindCollection())
        for accessing the gathered chunks.  However, it is not a good
        idea to keep a filehandle OR the clipboard open.  While
        a clipboard unit is open, no other applications can clip
        to the unit.  And while a file is open, you can't write the
        file back out.  So, instead of keeping the file or unit
        open, you can use copychunks (in copychunks.c) to create
        a copy of your gathered chunks, and do an early closeifile()
        (parse.c).  Then access and later write back out (if you wish)
        and deallocate your copied chunks via the routines in the
        copychunks module (findchunk, writechunklist, freechunklist).

WARNING REGARDING COMPLEX FORMS
        Regarding Complex FORMs - The parse.c module will enter complex
        formats such as CATSs, LISTs, and nested FORMs to find the FORM
        that you are interested in.  This is great.  However, if you are
        a read-modify-write program, you should warn your user when this
        occurs unless YOU are capable of recreating the complex format.
        Otherwise, your user may unknowingly destroy his complex file
        by writing over it with your program.  Example - a paint
        program could read an ILBM out of a complex LIST containing
        pictures and music, and then save it back out as a simple ILBM,
        causing the user to lose his music and other pictures.
        To determine if a complex form was entered after a load,
        check the (form)Info.ParseInfo.hunt field.  If TRUE (non-zero),
        then your file was found inside a complex format.

COMPILATION NOTES
        These modules and examples have been compiled using SAS C 5.10a
        and Manx C 5.0d, with 2.0 (37.1) include files and 2.0 amiga.lib.
        You must have at least 37.1 alib_protos.h (older versions of
        this include file contained the amiga.lib stdio protos also
        which conflict with both SAS and Manx stdio).  For Manx, I
        also had to comment out the conditional definition of abs() in
        libraries/mathffp.h.  These modules do not use mathffp, but
        the mathffp include file is pulled in by alib_protos.h.
        When compiling with Manx, a warning seems to be generated for
        each string constant assigned to a UBYTE * field, and also
        by some references to ilbm->colortable.



                LIST OF IFFP MODULES AND APPLICATIONS
                =====================================
        NOTE - Some useful functions are listed with each module
           See module source code for docs on each function.


APPLICATIONS (these require linkage with modules - see Makefiles)
============
ILBMDemo        Displays an ILBM, loads a brush, saves an ILBM, opt. print
ILBMLoad        Queries an ILBM and loads it into an existing screen
ILBMtoC         Outputs an ILBM as C source code
ILBMtoRaw       Converts an ILBM to raw plane/color file
RawtoILBM       Converts raw plane/color file (from ILBMtoRaw) to an ILBM
24bitDemo       Saves a simple 24-bit ILBM and then shows it 4 planes at
                a time (if given filename, just does the show part)
Play8SVX        Reads and plays an 8SVX sound effect or instrument
                  - LoadSample, UnloadSample, PlaySample, OpenAudio,
                    CloseAudio, and body load/unpack functions
ScreenSave      Save the front screen as an ILBM, with an icon
```

```
OTHER EXAMPLES (use iffparse.library directly and require no modules)
==============
Sift            Checks and prints outline of any IFF file (uses RAWSTEP)
ILBMScan        Prints out useful info about any ILBM
ClipFTXT        Demonstrates simply clipping of FTXT to/from clipboard
cycvb.c         Dan Silva's routine for interrupt based color cycling
apack.asm       Dr. Gerald Hull's assembler replacement for packer.c


GENERAL IFFPARSE SUPPORT MODULE
===============================
parse.c         File/clipboard IO and general parsing
                  - openifile, closeifile, parseifile, getcontext,
                    nextcontext, contextis, currentchunkis, PutCk chunk
                    writing function, and IFFerr text error routine


ILBM READ MODULES
=================
loadilbm.c      High level ILBM load routines which are passed filenames
                (calls getbitmap)
                  - loadbrush/unloadbrush, loadilbm/unloadilbm, and queryilbm
getbitmap.c     brush/bitmap loading (non-display, calls ilbmr.c)
                  - createbrush/deletebrush, getbitmap/freebitmap
getdisplay.c    bitmap load/display (calls screen.c, ilbmr.c)
                  - showilbm/unshowilbm, createdisplay/deletedisplay
screen.c        1.3/2.0 ECS/non-ECS compatible screen/window module
                  - opendisplay, openidscreen, modefallback, clipit
ilbmr.c         Lower level ILBM body/color load routines (calls unpacker.c)
                  - loadbody, loadcmap, getcolors/freecolors,
                    alloccolortable, getcamg (gets or creates modeid)
unpacker.c      BODY unpacker


ILBM WRITE MODULES
==================
saveilbm.c      High level ILBM saving routines which are passed filenames
                (calls ilbmw.c)
                  - screensave and saveilbm
ilbmw.c         Lower level ILBM body/color save routines (calls packer.c)
                  - InitBMHD, PutCMAP, PutBODY
packer.c        BODY packer


EXTRA MODULES
=============
copychunks.c    Chunk cloning and chunk list writing routines
                  - copychunks, findchunk, writechunklist, freechunklist
screendump.c    Screen printing module (iffparse not required)
bmprintc.c      Module to output ILBM as C code


INCLUDE FILES
=============
iffp/#?.h       This subdirectory may be kept in your current directory
                or in your main include directory.


Thanks to Steve Walton for his code changes for Manx/SAS compatibility,
and to Bill Barton and John Bittner for their comments and suggestions.
```

```
#MYLIBS= LIB:debug.lib

CC = lc
ASM = asm

CFLAGS = -cfistq -v -j73 -iINCLUDE:
AFLAGS = -iINCLUDE:
LFLAGS = SC BATCH ND

M      = modules/
A      = apps/

# Our iffparse support object modules to link with
IFFO      = $(M)parse.o $(M)Hook.o
ILBMRO    = $(M)ilbmr.o $(M)unpacker.o
ILBMSO    = $(M)getdisplay.o $(M)screen.o
ILBMLO    = $(M)loadilbm.o $(M)getbitmap.o
ILBMWO    = $(M)saveilbm.o $(M)ilbmw.o $(M)packer.o
ILBMO     = $(IFFO) $(ILBMRO) $(ILBMLO) $(ILBMSO) $(ILBMWO)
EXTRAO    = $(M)copychunks.o $(M)screendump.o $(M)bmprintc.o


# Our iffparse applications
APP1   = $(A)ILBMDemo/ILBMDemo
APP2   = $(A)ILBMLoad/ILBMLoad
APP3   = $(A)Play8SVX/Play8SVX
APP4   = $(A)ILBMtoC/ILBMtoC
APP5   = $(A)ILBMtoRaw/ILBMtoRaw
APP6   = $(A)ScreenSave/ScreenSave
APP7   = $(A)RawtoILBM/RawtoILBM
APP8   = $(A)24bitDemo/24bitDemo


# The object modules needed by each application example
APP1O  = $(APP1).o $(ILBMO) $(M)screendump.o $(M)copychunks.o
APP2O  = $(APP2).o $(IFFO) $(ILBMRO) $(ILBMLO) $(ILBMSO)
APP3O  = $(APP3).o $(IFFO)
APP4O  = $(APP4).o $(IFFO) $(ILBMRO) $(ILBMLO) $(M)bmprintc.o
APP5O  = $(APP5).o $(IFFO) $(ILBMRO) $(ILBMLO)
APP6O  = $(APP6).o $(IFFO) $(ILBMWO)
APP7O  = $(APP7).o $(IFFO) $(ILBMWO)
APP8O  = $(APP8).o $(IFFO) $(ILBMRO) $(ILBMLO) $(ILBMSO) $(ILBMWO)

.SUFFIXES:
.SUFFIXES:      .o .c .h .asm .i

# Make all of the applications
all:    $(APP1) $(APP2) $(APP3) $(APP4) $(APP5) $(APP6) $(APP7) $(APP8)

# Linkage for each application

$(APP1): $(APP1O)
  blink <WITH <
FROM lib:c.o $(APP1O)
LIBRARY lib:lc.lib LIB:amiga.lib $(MYLIBS)
TO $(APP1) $(LFLAGS)
<

$(APP2): $(APP2O)
  blink <WITH <
FROM lib:c.o $(APP2O)
LIBRARY lib:lc.lib LIB:amiga.lib $(MYLIBS)
TO $(APP2) $(LFLAGS)
<
```

Makefile.SAS

```
$(APP3): $(APP30)
  blink <WITH <
FROM lib:c.o $(APP30)
LIBRARY lib:lc.lib LIB:amiga.lib $(MYLIBS)
TO $(APP3) $(LFLAGS)
<

$(APP4): $(APP40)
  blink <WITH <
FROM lib:c.o $(APP40)
LIBRARY lib:lc.lib LIB:amiga.lib $(MYLIBS)
TO $(APP4) $(LFLAGS)
<

$(APP5): $(APP50)
  blink <WITH <
FROM lib:c.o $(APP50)
LIBRARY lib:lc.lib LIB:amiga.lib $(MYLIBS)
TO $(APP5) $(LFLAGS)
<

$(APP6): $(APP60)
  blink <WITH <
FROM lib:c.o $(APP60)
LIBRARY lib:lc.lib LIB:amiga.lib $(MYLIBS)
TO $(APP6) $(LFLAGS)
<

$(APP7): $(APP70)
  blink <WITH <
FROM lib:c.o $(APP70)
LIBRARY lib:lc.lib LIB:amiga.lib $(MYLIBS)
TO $(APP7) $(LFLAGS)
<

$(APP8): $(APP80)
  blink <WITH <
FROM lib:c.o $(APP80)
LIBRARY lib:lc.lib LIB:amiga.lib $(MYLIBS)
TO $(APP8) $(LFLAGS)
<

.c.o:
        $(CC) $(CFLAGS) $*.c

.asm.o:
        $(ASM) $(AFLAGS) $*.asm
```

Makefile.Manx

```
#MYLIBS= LIB:debug.lib

CC = cc
ASM = as

CFLAGS = -IWork:manxinclude
AFLAGS =
LFLAGS =

M       = modules/
A       = apps/

# Our iffparse support object modules to link with
IFFO      = $(M)parse.o $(M)Hook.o
ILBMRO    = $(M)ilbmr.o $(M)unpacker.o
ILBMSO    = $(M)getdisplay.o $(M)screen.o
ILBMLO    = $(M)loadilbm.o $(M)getbitmap.o
ILBMWO    = $(M)saveilbm.o $(M)ilbmw.o $(M)packer.o
ILBMO     = $(IFFO) $(ILBMRO) $(ILBMLO) $(ILBMSO) $(ILBMWO)
EXTRAO    = $(M)copychunks.o $(M)screendump.o $(M)bmprintc.o

# Our iffparse applications
APP1   = $(A)ILBMDemo/ILBMDemo
APP2   = $(A)ILBMLoad/ILBMLoad
APP3   = $(A)Play8SVX/Play8SVX
APP4   = $(A)ILBMtoC/ILBMtoC
APP5   = $(A)ILBMtoRaw/ILBMtoRaw
APP6   = $(A)ScreenSave/ScreenSave
APP7   = $(A)RawtoILBM/RawtoILBM
APP8   = $(A)24bitDemo/24bitDemo

# The object modules needed by each application example
APP10  = $(APP1).o $(ILBMO) $(M)screendump.o $(M)copychunks.o
APP20  = $(APP2).o $(IFFO) $(ILBMRO) $(ILBMLO) $(ILBMSO)
APP30  = $(APP3).o $(IFFO)
APP40  = $(APP4).o $(IFFO) $(ILBMRO) $(ILBMLO) $(M)bmprintc.o
APP50  = $(APP5).o $(IFFO) $(ILBMRO) $(ILBMLO)
APP60  = $(APP6).o $(IFFO) $(ILBMWO)
APP70  = $(APP7).o $(IFFO) $(ILBMWO)
APP80  = $(APP8).o $(IFFO) $(ILBMRO) $(ILBMLO) $(ILBMSO) $(ILBMWO)

.SUFFIXES:
.SUFFIXES:      .o .c .h .asm .i


# Make all of the applications
all:    $(APP1) $(APP2) $(APP3) $(APP4) $(APP5) $(APP6) $(APP7) $(APP8)

# Linkage for each application

$(APP1): $(APP10)
        ln -o $(APP1) $(LFLAGS) $(APP10) -lc +l amiga.lib

$(APP2): $(APP20)
        ln -o $(APP2) $(LFLAGS) $(APP20) -lc +l amiga.lib

$(APP3): $(APP30)
        ln -o $(APP3) $(LFLAGS) $(APP30) -lc +l amiga.lib

$(APP4): $(APP40)
        ln -o $(APP4) $(LFLAGS) $(APP40) -lc +l amiga.lib

$(APP5): $(APP50)
        ln -o $(APP5) $(LFLAGS) $(APP50) -lc +l amiga.lib
```

Makefile.Manx

```
$(APP6): $(APP60)
	ln -o $(APP6) $(LFLAGS) $(APP60) -lc +l amiga.lib

$(APP7): $(APP70)
	ln -o $(APP7) $(LFLAGS) $(APP70) -lc +l amiga.lib

$(APP8): $(APP80)
	ln -o $(APP8) $(LFLAGS) $(APP80) -lc +l amiga.lib


.c.o:
	cc $(CFLAGS) -o $*.o $*.c

.asm.o:
	as $(AFLAGS) -o $*.o $*.asm
```

iffp/8svx.h

```
/*----------------------------------------------------------------------*
 * 8SVX.H  Definitions for 8-bit sampled voice (VOX).   2/10/86
 *
 * By Jerry Morrison and Steve Hayes, Electronic Arts.
 * This software is in the public domain.
 *
 * Modified for use with iffparse.library 05/91 - CAS_CBM
 *
 * This version for the Commodore-Amiga computer.
 *----------------------------------------------------------------------*/
#ifndef EIGHTSVX_H
#define EIGHTSVX_H

#ifndef COMPILER_H
#include "iffp/compiler.h"
#endif

#include "iffp/iff.h"

#define ID_8SVX      MAKE_ID('8', 'S', 'V', 'X')
#define ID_VHDR      MAKE_ID('V', 'H', 'D', 'R')

#define ID_ATAK      MAKE_ID('A', 'T', 'A', 'K')
#define ID_RLSE      MAKE_ID('R', 'L', 'S', 'E')

/* defined in iffp/iff.h
#define ID_NAME      MAKE_ID('N', 'A', 'M', 'E')
#define ID_Copyright MAKE_ID('(', 'c', ')', ' ')
#define ID_AUTH      MAKE_ID('A', 'U', 'T', 'H')
#define ID_ANNO      MAKE_ID('A', 'N', 'N', 'O')
#define ID_BODY      MAKE_ID('B', 'O', 'D', 'Y')
*/


/* ---------- Voice8Header ---------------------------------------------*/
typedef LONG Fixed;     /* A fixed-point value, 16 bits to the left of
                         * the point and 16 to the right. A Fixed is a
                         * number of 2**16ths, i.e. 65536ths. */
#define Unity 0x10000L  /* Unity = Fixed 1.0 = maximum volume */

/* sCompression: Choice of compression algorithm applied to the samples. */
#define sCmpNone       0        /* not compressed */
#define sCmpFibDelta   1        /* Fibonacci-delta encoding (Appendix C) */
                                /* Could be more kinds in the future. */
typedef struct {
    ULONG oneShotHiSamples,     /* # samples in the high octave 1-shot part */
          repeatHiSamples,      /* # samples in the high octave repeat part */
          samplesPerHiCycle;    /* # samples/cycle in high octave, else 0 */
    UWORD samplesPerSec;        /* data sampling rate */
    UBYTE ctOctave,             /* # of octaves of waveforms */
          sCompression;         /* data compression technique used */
    Fixed volume;               /* playback nominal volume from 0 to Unity
                                 * (full volume). Map this value into
                                 * the output hardware's dynamic range.
                                 */
    } Voice8Header;

/* ---------- NAME -----------------------------------------------------*/
/* NAME chunk contains a CHAR[], the voice's name. */

/* ---------- Copyright ------------------------------------------------*/
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice. */

/* ---------- AUTH -----------------------------------------------------*/
/* AUTH chunk contains a CHAR[], the author's name. */
```

iffp/8svx.h

```
/* ---------- ANNO --------------------------------------------------------*/
/* ANNO chunk contains a CHAR[], the author's text annotations. */

/* ---------- Envelope ATAK & RLSE ------------------------------------*/
typedef struct {
    UWORD duration;     /* segment duration in milliseconds, > 0 */
    Fixed dest;         /* destination volume factor */
    } EGPoint;

/* ATAK and RLSE chunks contain an EGPoint[], piecewise-linear envelope. */

/* The envelope defines a function of time returning Fixed values.
 * It's used to scale the nominal volume specified in the Voice8Header.
 */

/* ---------- BODY --------------------------------------------------------*/
/* BODY chunk contains a BYTE[], array of audio data samples. */
/* (8-bit signed numbers, -128 through 127.) */


/* ---------- 8SVX Writer Support Routines -----------------------------*/

/* Just call this macro to write a VHDR chunk. */
#define PutVHDR(iff, vHdr)  \
    PutCk(iff, ID_VHDR, sizeof(Voice8Header), (BYTE *)vHdr)

#endif
```

iffp/8svxapp.h

```
/* 8svxapp.h
 * - definition of EightSVXInfo structure
 * - inclusion of includes needed by modules and application
 * - application-specific definitions
 */
#ifndef EIGHTSVXAPP_H
#define EIGHTSVXAPP_H

#include "iffp/8svx.h"

#include <devices/audio.h>

#define MAXOCT 16

struct EightSVXInfo {
        /* general parse.c related */
        struct  ParseInfo ParseInfo;

        /* For convenient access to VHDR, Name, and sample.
         * Other chunks will be accessible through FindProp()
         *  (or findchunk() if the chunks have been copied)
         */
        /* 8SVX */
        Voice8Header    Vhdr;

        BYTE            *sample;
        ULONG           samplebytes;

        BYTE            *osamps[MAXOCT];
        ULONG           osizes[MAXOCT];
        BYTE            *rsamps[MAXOCT];
        ULONG           rsizes[MAXOCT];
        ULONG           spcycs[MAXOCT];

        UBYTE           name[80];

        ULONG           Reserved[8];    /* must be 0 for now */

        /* Applications may add variables here */
        };

/* referenced by modules */
extern struct Library *IFFParseBase;

#endif
```

iffp/amiga.h

```
/* IFF application include files
 */

#ifndef  AMIGA_H
#define  AMIGA_H

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/libraries.h>

#include <libraries/dos.h>

#include <intuition/intuition.h>
#include <intuition/screens.h>

#include <graphics/view.h>
#include <graphics/displayinfo.h>
#include <graphics/videocontrol.h>
#include <graphics/gfxmacros.h>

#include <libraries/iffparse.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/iffparse_protos.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "iffp/debug.h"

#endif
```

iffp/compiler.h

```
#ifndef COMPILER_H
#define COMPILER_H
/*** compiler.h ************************************************************/
/*  Steve Shaw                                                      1/29/86 */
/*  Portability file to handle compiler idiosyncrasies.                     */
/*                                                                          */
/* This software is in the public domain.                                   */
/* modified 05/91 for use with iffparse - CAS_CBM                           */
/***************************************************************************/

#ifndef EXEC_TYPES_H
#include "exec/types.h"
#endif

/*
#define NO_PROTOS
*/

#endif COMPILER_H
```

iffp/debug.h Page 1

```
/*
 * mydebug.h - #include this file sometime after stdio.h
 * Set MYDEBUG to 1 to turn on debugging, 0 to turn off debugging
 */

#ifndef MYDEBUG_H
#define MYDEBUG_H

#define MYDEBUG  0


#if MYDEBUG

/*
 * MYDEBUG User Options
 */

/* Set to 1 to turn second level D2(bug()) statements */
#define DEBUGLEVEL2     1

/* Set to a non-zero # of ticks if a delay is wanted after each debug message */
#define DEBUGDELAY              0

/* Always non-zero for the DDx macros */
#define DDEBUGDELAY             50

/* Set to 1 for serial debugging (link with debug.lib) */
#define KDEBUG          0

/* Set to 1 for parallel debugging (link with ddebug.lib) */
#define DDEBUG          0

#endif /* MYDEBUG */


/* Prototypes for Delay, kprintf, dprintf. Or use proto/dos.h or functions.h. */
#include <clib/dos_protos.h>
void kprintf(UBYTE *fmt,...);
void dprintf(UBYTE *fmt,...);

/*
 * D(bug()), D2(bug()), DQ((bug()) only generate code if MYDEBUG is non-zero
 *
 * Use D(bug()) for general debugging, D2(bug()) for extra debugging that
 * you usually won't need to see, DD(bug()) for debugging statements that
 * you always want followed by a delay, and DQ(bug()) for debugging that
 * you'll NEVER want a delay after (ie. debugging inside a Forbid, Disable,
 * Task, or Interrupt)
 *
 * Some example uses (all are used the same):
 * D(bug("about to do xyz. variable = $%lx\n",myvariable));
 * D2(bug("v1=$%lx v2=$%lx v3=$%lx\n",v1,v2,v3));
 * DQ(bug("in subtask: variable = $%lx\n",myvariable));
 * DD(bug("About to do xxx\n"));
 *
 * Set MYDEBUG above to 1 when debugging is desired and recompile the modules
 *  you wish to debug.  Set to 0 and recompile to turn off debugging.
 *
 * User options set above:
 * Set DEBUGDELAY to a non-zero # of ticks (ex. 50) when a delay is desired.
 * Set DEBUGLEVEL2 nonzero to turn on second level (D2) debugging statements
 * Set KDEBUG to 1 and link with debug.lib for serial debugging.
 * Set DDEBUG to 1 and link with ddebug.lib for parallel debugging.
 */
```

iffp/debug.h Page 2

```
/*
 * Debugging function automaticaly set to printf, kprintf, or dprintf
 */

#if KDEBUG
#define bug kprintf
#elif DDEBUG
#define bug dprintf
#else   /* else changes all bug's to printf's */
#define bug printf
#endif

/*
 * Debugging macros
 */

/* D(bug(        delays DEBUGDELAY if DEBUGDELAY is > 0
 * DD(bug(       always delays DDEBUGDELAY
 * DQ(bug(       (debug quick) never uses Delay.  Use in forbids,disables,ints
 * The similar macros with "2" in their names are second level debugging
 */
#if MYDEBUG    /* Turn on first level debugging */
#define D(x)   (x); if(DEBUGDELAY>0) Delay(DEBUGDELAY)
#define DD(x)  (x); Delay(DDEBUGDELAY)
#define DQ(x)  (x)
#if DEBUGLEVEL2 /* Turn on second level debugging */
#define D2(x)   (x); if(DEBUGDELAY>0) Delay(DEBUGDELAY)
#define DD2(x)  (x); Delay(DDEBUGDELAY)
#define DQ2(x)  (x)
#else  /* Second level debugging turned off */
#define D2(x) ;
#define DD2(x) ;
#define DQ2(x) ;
#endif /* DEBUGLEVEL2 */
#else  /* First level debugging turned off */
#define D(x) ;
#define DQ(x) ;
#define D2(x) ;
#define DD(x) ;
#endif


#endif /* MYDEBUG_H */
```

```
/*
 *
 * iff.h:       General Definitions for IFFParse modules
 *
 * 6/27/91
 */

#ifndef IFFP_IFF_H
#define IFFP_IFF_H

#include "iffp/compiler.h"

#ifndef EXEC_TYPES_H
#include <exec/types.h>
#endif
#ifndef EXEC_MEMORY_H
#include <exec/memory.h>
#endif
#ifndef UTILITY_TAGITEM_H
#include <utility/tagitem.h>
#endif
#ifndef UTILITY_HOOKS_H
#include <utility/hooks.h>
#endif
#ifndef LIBRARIES_IFFPARSE_H
#include <libraries/iffparse.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef MYDEBUG_H
#include "iffp/debug.h"
#endif

#ifndef NO_PROTOS
#include <clib/exec_protos.h>
#include <clib/iffparse_protos.h>
#endif

#ifndef MAX
#define MAX(a,b)        ((a) > (b) ? (a) : (b))
#endif
#ifndef MIN
#define MIN(a,b)        ((a) < (b) ? (a) : (b))
#endif
#ifndef ABS
#define ABS(x)          ((x) < 0 ? -(x) : (x))
#endif


#define CkErr(expression)  {if (!error) error = (expression);}
#define ChunkMoreBytes(cn)      (cn->cn_Size - cn->cn_Scan)
#define IS_ODD(a)               (a & 1)

#define IFF_OKAY        0L
#define CLIENT_ERROR    1L
#define NOFILE          5L

#define message printf

/* Generic Chunk ID's we may encounter */
#define ID_ANNO         MAKE_ID('A','N','N','O')
#define ID_AUTH         MAKE_ID('A','U','T','H')
#define ID_CHRS         MAKE_ID('C','H','R','S')
```

```
#define ID_Copyright    MAKE_ID('(','c',')',' ')
#define ID_CSET         MAKE_ID('C','S','E','T')
#define ID_FVER         MAKE_ID('F','V','E','R')
#define ID_NAME         MAKE_ID('N','A','M','E')
#define ID_TEXT         MAKE_ID('T','E','X','T')
#define ID_BODY         MAKE_ID('B','O','D','Y')


/* Used to keep track of allocated IFFHandle, and whether file is
 * clipboard or not, filename, copied chunks, etc.
 * This structure is included in the beginning of every
 * form-specific info structure used by the example modules.
 */
struct ParseInfo {
        /* general parse.c related */
        struct  IFFHandle *iff;         /* to be alloc'd with AllocIFF */
        UBYTE   *filename;              /* current filename of this ui */
        LONG    *propchks;              /* properties to get */
        LONG    *collectchks;           /* properties to collect */
        LONG    *stopchks;              /* stop on these (like BODY) */
        BOOL    opened;                 /* this iff has been opened */
        BOOL    clipboard;              /* file is clipboard */
        BOOL    hunt;                   /* we are parsing a complex file */
        BOOL    Reserved1;              /* must be zero for now */

        /* for copychunks.c - for read/modify/write programs
         * and programs that need to keep parsed chunk info
         * around after closing file.
         * Deallocated by freechunklist();
         */
        struct Chunk *copiedchunks;

        /* application may hang its own list of new chunks here
         * just to keep it with the frame.
         */
        struct Chunk *newchunks;

        ULONG   Reserved[8];
        };


/*
 * Used by some modules to save or pass a singly linked list of chunks
 */
struct Chunk {
        struct  Chunk *ch_Next;
        long    ch_Type;
        long    ch_ID;
        long    ch_Size;
        void    *ch_Data;
};


#ifndef NO_PROTOS
/* parse.c */
LONG openifile(struct ParseInfo *,UBYTE *,ULONG);
void closeifile(struct ParseInfo *);
LONG parseifile(struct ParseInfo *,
                LONG, LONG, LONG *, LONG *, LONG *);
LONG getcontext(struct IFFHandle *);
LONG nextcontext(struct IFFHandle *);
LONG currentchunkis(struct IFFHandle *, LONG type, LONG id);
LONG contextis(struct IFFHandle *, LONG type, LONG id);
UBYTE *findpropdata(struct IFFHandle *iff, LONG type, LONG id);
void initiffasstdio(struct IFFHandle *);
UBYTE *IFFerr(LONG);
```

```
LONG chkcnt(LONG *);
long PutCk(struct IFFHandle *iff, long id, long size, void *data);

/* copychunks.c */
struct Chunk *copychunks(struct IFFHandle *iff,
                LONG *propchks, LONG *collectchks, ULONG memtype);
void freechunklist(struct Chunk *first);
struct Chunk *findchunk(struct Chunk *first, long type, long id);
long writechunklist(struct IFFHandle *iff, struct Chunk *first);
#endif /* NO_PROTOS */

#endif /* IFFP_IFF_H */
```

```
/*
 *
 * ilbm.h:      Definitions for IFFParse ILBM reader.
 *
 * 6/27/91
 */

#ifndef IFFP_ILBM_H
#define IFFP_ILBM_H

#ifndef IFFP_IFF_H
#include "iffp/iff.h"
#endif

#ifndef INTUITION_INTUITION_H
#include <intuition/intuition.h>
#endif
#ifndef GRAPHICS_VIDEOCONTROL_H
#include <graphics/videocontrol.h>
#endif

#ifndef NO_PROTOS
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>
#include <clib/alib_protos.h>
#endif

/*  IFF types we may encounter  */
#define ID_ILBM         MAKE_ID('I','L','B','M')

/* ILBM Chunk ID's we may encounter
 * (see iffp/iff.h for some other generic chunks)
 */
#define ID_BMHD         MAKE_ID('B','M','H','D')
#define ID_CMAP         MAKE_ID('C','M','A','P')
#define ID_CRNG         MAKE_ID('C','R','N','G')
#define ID_CCRT         MAKE_ID('C','C','R','T')
#define ID_GRAB         MAKE_ID('G','R','A','B')
#define ID_SPRT         MAKE_ID('S','P','R','T')
#define ID_DEST         MAKE_ID('D','E','S','T')
#define ID_CAMG         MAKE_ID('C','A','M','G')

/* Use this constant instead of sizeof(ColorRegister). */
#define sizeofColorRegister  3

typedef WORD Color4;   /* Amiga RAM version of a color-register,
          * with 4 bits each RGB in low 12 bits.*/

/* Maximum number of bitplanes storable in BitMap structure */
#define MAXAMDEPTH 8
#define MAXAMCOLORREG 32

/* Maximum planes we can save */
#define MAXSAVEDEPTH 24

/* convert width to BytesPerRow */
#define BytesPerRow(w)  ((w) + 15 >> 4 << 1)
#define BitsPerRow(w)   ((w) + 15 >> 4 << 4)

/* Flags that should be masked out of old 16-bit CAMG before save or use.
 * Note that 32-bit mode id (non-zero high word) bits should not be twiddled
 */
#define BADFLAGS  (SPRITES|VP_HIDE|GENLOCK_AUDIO|GENLOCK_VIDEO)
#define OLDCAMGMASK  (~BADFLAGS)
```

```
/*  Masking techniques  */
#define mskNone                  0
#define mskHasMask               1
#define mskHasTransparentColor   2
#define mskLasso                 3

/*  Compression techniques  */
#define cmpNone                  0
#define cmpByteRun1              1

#define RowBytes(w)     ((((w) + 15) >> 4) << 1)

/* ---------- BitMapHeader ---------------------------------------------*/
/*  Required Bitmap header (BMHD) structure describes an ILBM */
typedef struct {
        UWORD   w, h;            /* Width, height in pixels */
        WORD    x, y;            /* x, y position for this bitmap  */
        UBYTE   nPlanes;         /* # of planes (not including mask) */
        UBYTE   masking;         /* a masking technique listed above */
        UBYTE   compression;     /* cmpNone or cmpByteRun1 */
        UBYTE   reserved1;       /* must be zero for now */
        UWORD   transparentColor;
        UBYTE   xAspect, yAspect;
        WORD    pageWidth, pageHeight;
} BitMapHeader;

/* ---------- ColorRegister --------------------------------------------*/
/* A CMAP chunk is a packed array of ColorRegisters (3 bytes each). */
typedef struct {
    UBYTE red, green, blue;   /* MUST be UBYTEs so ">> 4" won't sign extend.*/
    } ColorRegister;

/* ---------- Point2D --------------------------------------------------*/
/* A Point2D is stored in a GRAB chunk. */
typedef struct {
    WORD x, y;      /* coordinates (pixels) */
    } Point2D;

/* ---------- DestMerge ------------------------------------------------*/
/* A DestMerge is stored in a DEST chunk. */
typedef struct {
    UBYTE depth;   /* # bitplanes in the original source */
    UBYTE pad1;      /* UNUSED; for consistency store 0 here */
    UWORD planePick;   /* how to scatter source bitplanes into destination */
    UWORD planeOnOff;   /* default bitplane data for planePick */
    UWORD planeMask;   /* selects which bitplanes to store into */
    } DestMerge;

/* ---------- SpritePrecedence -----------------------------------------*/
/* A SpritePrecedence is stored in a SPRT chunk. */
typedef UWORD SpritePrecedence;

/* ---------- Camg Amiga Viewport Mode ---------------------------------*/
/* A Commodore Amiga ViewPort->Modes is stored in a CAMG chunk. */
/* The chunk's content is declared as a LONG. */
typedef struct {
   ULONG ViewModes;
   } CamgChunk;

/* ---------- CRange cycling chunk -------------------------------------*/
#define RNG_NORATE  36   /* Dpaint uses this rate to mean non-active */
/* A CRange is store in a CRNG chunk. */
typedef struct {
    WORD  pad1;              /* reserved for future use; store 0 here */
    WORD  rate;      /* 60/sec=16384, 30/sec=8192, 1/sec=16384/60=273 */
    WORD  active;     /* bit0 set = active, bit 1 set = reverse */
```

```
    UBYTE low, high;   /* lower and upper color registers selected */
    } CRange;

/* ---------- Ccrt (Graphicraft) cycling chunk -------------------------*/
/* A Ccrt is stored in a CCRT chunk. */
typedef struct {
   WORD  direction;  /* 0=don't cycle, 1=forward, -1=backwards */
   UBYTE start;      /* range lower */
   UBYTE end;        /* range upper */
   LONG  seconds;    /* seconds between cycling */
   LONG  microseconds; /* msecs between cycling */
   WORD  pad;        /* future exp - store 0 here */
   } CcrtChunk;

/* If you are writing all of your chunks by hand,
 * you can use these macros for these simple chunks.
 */
#define putbmhd(iff, bmHdr)  \
    PutCk(iff, ID_BMHD, sizeof(BitMapHeader), (BYTE *)bmHdr)
#define putgrab(iff, point2D)  \
    PutCk(iff, ID_GRAB, sizeof(Point2D), (BYTE *)point2D)
#define putdest(iff, destMerge)  \
    PutCk(iff, ID_DEST, sizeof(DestMerge), (BYTE *)destMerge)
#define putsprt(iff, spritePrec)  \
    PutCk(iff, ID_SPRT, sizeof(SpritePrecedence), (BYTE *)spritePrec)
#define putcamg(iff, camg)  \
    PutCk(iff, ID_CAMG, sizeof(CamgChunk),(BYTE *)camg)
#define putcrng(iff, crng)  \
    PutCk(iff, ID_CRNG, sizeof(CRange),(BYTE *)crng)
#define putccrt(iff, ccrt)  \
    PutCk(iff, ID_CCRT, sizeof(CcrtChunk),(BYTE *)ccrt)

#ifndef NO_PROTOS
/* unpacker.c */
BOOL unpackrow(BYTE **pSource, BYTE **pDest, WORD srcBytes0, WORD dstBytes0);

/* packer.c */
LONG packrow(BYTE **pSource, BYTE **pDest, LONG rowSize);

/* ilbmr.c  ILBM reader routines */
LONG loadbody(struct IFFHandle *iff, struct BitMap *bitmap,
                BitMapHeader *bmhd);
LONG loadbody2(struct IFFHandle *iff, struct BitMap *bitmap,
                BYTE *mask, BitMapHeader *bmhd,
                BYTE *buffer, ULONG bufsize);
LONG loadcmap(struct IFFHandle *, WORD *colortable, USHORT *pNcolors);
LONG getcolors(struct ILBMInfo *ilbm);
void freecolors(struct ILBMInfo *ilbm);
LONG alloccolortable(struct ILBMInfo *ilbm);
ULONG getcamg(struct ILBMInfo *ilbm);

/* ilbmw.c  ILBM writer routines */
long initbmhd(BitMapHeader *bmhd, struct BitMap *bitmap,
              WORD masking, WORD compression, WORD transparentColor,
              WORD width, WORD height, WORD pageWidth, WORD pageHeight,
              ULONG modeid);
long putcmap(struct IFFHandle *iff,APTR colortable,UWORD ncolors,UWORD bitspergun);
long putbody(struct IFFHandle *iff, struct BitMap *bitmap,
                BYTE *mask, BitMapHeader *bmHdr,
                BYTE *buffer, LONG bufsize);

/* getdisplay.c (used to load a display) */
LONG showilbm(struct ILBMInfo *ilbm, UBYTE *filename);
void unshowilbm(struct ILBMInfo *ilbm);
LONG createdisplay(struct ILBMInfo *);
void deletedisplay(struct ILBMInfo *);
```

```
LONG getdisplay(struct ILBMInfo *);
void freedisplay(struct ILBMInfo *);

/* getbitmap.c (used if just loading brush or bitmap) */
LONG createbrush(struct ILBMInfo *);
void deletebrush(struct ILBMInfo *);
LONG getbitmap(struct ILBMInfo *);
void freebitmap(struct ILBMInfo *);

/* screen.c (opens 1.3 or 2.0 screen) */
struct Screen *openidscreen(struct ILBMInfo *,SHORT,SHORT,SHORT,ULONG);
struct Window *opendisplay(struct ILBMInfo *,SHORT,SHORT,SHORT,ULONG);
ULONG  modefallback(ULONG, SHORT, SHORT, SHORT);
void clipit(SHORT wide, SHORT high, struct Rectangle *spos,
        struct Rectangle *dclip, struct Rectangle *txto,
        struct Rectangle *stdo,struct Rectangle *maxo,
        struct Rectangle * uclip);
void closedisplay(struct ILBMInfo *);
void modeErrorMsg(ULONG,ULONG);

/* loadilbm.c */
LONG loadbrush(struct ILBMInfo *ilbm, UBYTE *filename);
void unloadbrush(struct ILBMInfo *ilbm);

LONG queryilbm(struct ILBMInfo *ilbm, UBYTE *filename);

LONG loadilbm(struct ILBMInfo *ilbm, UBYTE *filename);
void unloadilbm(struct ILBMInfo *ilbm);

/* saveilbm.c */
LONG screensave(struct ILBMInfo *ilbm,
                        struct Screen *scr,
                        struct Chunk *chunklist1, struct Chunk *chunklist2,
                        UBYTE *filename);

LONG saveilbm(struct ILBMInfo *ilbm,
                struct BitMap *bitmap, ULONG modeid,
                WORD width, WORD height, WORD pagewidth, WORD pageheight,
                APTR colortable, UWORD count, UWORD bitspergun,
                WORD masking, WORD transparentColor,
                struct Chunk *chunklist1, struct Chunk *chunklist2,
                UBYTE *filename);


/* screendump.c (print screen or brush) */
int screendump(struct Screen *scr,
     UWORD srcx, UWORD srcy, UWORD srcw, UWORD srch,
     LONG destcols, UWORD special);

/* bmprintc.c (write C source for ILBM) */
void BMPrintCRep(struct BitMap *bm, FILE *fp, UBYTE *name, UBYTE *fmt);

#endif /* NO_PROTOS */

#endif /* IFFP_ILBM_H */
```

```
/* ilbmapp.h
 * - definition of ILBMInfo structure
 * - inclusion of includes needed by modules and application
 * - application-specific definitions
 *
 * 07/03/91 - added ilbm->stags for screen.c
 */
#ifndef ILBMAPP_H
#define ILBMAPP_H

#include "iffp/ilbm.h"

struct ILBMInfo {
        /* general parse.c related */
        struct  ParseInfo ParseInfo;

        /* The following variables are for
         * programs using the ILBM-related modules.
         * They may be removed or replaced for
         * programs parsing other forms.
         */
        /* ILBM */
        BitMapHeader Bmhd;              /* filled in by load and save ops */
        ULONG   camg;                   /* filled in by load and save ops */
        Color4  *colortable;            /* allocated by getcolors */
        ULONG   ctabsize;               /* size of colortable in bytes */
        USHORT  ncolors;                /* number of color registers loaded */
        USHORT  Reserved1;

        /* for getbitmap.c */
        struct BitMap *brbitmap;        /* for loaded brushes only */

        /* for screen.c */
        struct Screen *scr;             /* screen of loaded display   */
        struct Window *win;             /* window of loaded display   */
        struct ViewPort *vp;            /* viewport of loaded display */
        struct RastPort *srp;           /* screen's rastport */
        struct RastPort *wrp;           /* window's rastport */
        BOOL TBState;                   /* state of titlebar hiddenness */

        /* caller preferences */
        struct NewWindow *windef;       /* definition for window */
        UBYTE *stitle;              /* screen title */
        LONG stype;                 /* additional screen types */
        WORD ucliptype;             /* overscan display clip type */
        BOOL EHB;                   /* default to EHB for 6-plane/NoCAMG */
        BOOL Video;                 /* Max Video Display Clip (non-adjustable) */
        BOOL Autoscroll;            /* Enable Autoscroll of screens */
        BOOL Notransb;              /* Borders not transparent to genlock */
        ULONG *stags;               /* Additional screen tags for 2.0 screens */

        ULONG Reserved[7];          /* must be 0 for now */

        /* Application-specific variables may go here */
        };

/* referenced by modules */

extern struct Library *IFFParseBase;

/* protos for application module(s) */

#endif
```

iffp/packer.h

```
#ifndef PACKER_H
#define PACKER_H
/*----------------------------------------------------------------------*
 * PACKER.H  typedefs for Data-Compresser.                      1/22/86
 *
 * This module implements the run compression algorithm "cmpByteRun1"; the
 * same encoding generated by Mac's PackBits.
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *----------------------------------------------------------------------*/

#include <exec/types.h>

/* This macro computes the worst case packed size of a "row" of bytes. */
#define MaxPackedSize(rowSize)  ( (rowSize) + ( ((rowSize)+127) >> 7 ) )


/* Given POINTERS to POINTER variables, packs one row, updating the source
 * and destination pointers. Returns the size in bytes of the packed row.
 * ASSUMES destination buffer is large enough for the packed row.
 * See MaxPackedSize. */
extern LONG PackRow(BYTE **, BYTE **, LONG);
                 /*  pSource, pDest,   rowSize */

/* Given POINTERS to POINTER variables, unpacks one row, updating the source
 * and destination pointers until it produces dstBytes bytes (i.e., the
 * rowSize that went into PackRow).
 * If it would exceed the source's limit srcBytes or if a run would overrun
 * the destination buffer size dstBytes, it stops and returns TRUE.
 * Otherwise, it returns FALSE (no error). */
extern BOOL UnPackRow(BYTE **, BYTE **, WORD,     WORD);
                   /*  pSource, pDest,   srcBytes, dstBytes  */


BYTE *PutDump(BYTE *, int);
BYTE *PutRun(BYTE *,int,int);
LONG PackRow(BYTE **,BYTE **,LONG);
BOOL UnPackRow(BYTE **,BYTE **,WORD,WORD);

#endif
```

iffp/smus.h

```
/*----------------------------------------------------------------------*
 * SMUS.H  Definitions for Simple MUSical score.   2/12/86
 *
 * By Jerry Morrison and Steve Hayes, Electronic Arts.
 * This software is in the public domain.
 *
 * Modified for use with iffparse.library 05/91 - CAS_CBM
 *
 * This version for the Commodore-Amiga computer.
 *----------------------------------------------------------------------*/

#ifndef SMUS_H
#define SMUS_H

#ifndef COMPILER_H
#include "iffp/compiler.h"
#endif

#include "iffp/iff.h"

#define ID_SMUS      MAKE_ID('S', 'M', 'U', 'S')
#define ID_SHDR      MAKE_ID('S', 'H', 'D', 'R')

/* Now defined in iffp/iff.h as generic chunks
#define ID_NAME      MAKE_ID('N', 'A', 'M', 'E')
#define ID_Copyright MAKE_ID('(', 'c', ')', ' ')
#define ID_AUTH      MAKE_ID('A', 'U', 'T', 'H')
#define ID_ANNO      MAKE_ID('A', 'N', 'N', 'O')
*/

#define ID_INS1      MAKE_ID('I', 'N', 'S', '1')
#define ID_TRAK      MAKE_ID('T', 'R', 'A', 'K')


/* ---------- SScoreHeader ---------------------------------------------*/
typedef struct {
    UWORD tempo;        /* tempo, 128ths quarter note/minute */
    UBYTE volume;       /* playback volume 0 through 127 */
    UBYTE ctTrack;      /* count of tracks in the score */
    } SScoreHeader;

/* ---------- NAME -----------------------------------------------------*/
/* NAME chunk contains a CHAR[], the musical score's name. */

/* ---------- Copyright (c) --------------------------------------------*/
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice. */

/* ---------- AUTH -----------------------------------------------------*/
/* AUTH chunk contains a CHAR[], the name of the score's author. */

/* ---------- ANNO -----------------------------------------------------*/
/* ANNO chunk contains a CHAR[], the author's text annotations. */

/* ---------- INS1 -----------------------------------------------------*/
/* Constants for the RefInstrument's "type" field. */
#define INS1_Name  0    /* just use the name; ignore data1, data2 */
#define INS1_MIDI  1    /* <data1, data2> = MIDI <channel, preset> */

typedef struct {
    UBYTE iRegister;    /* set this instrument register number */
    UBYTE type;         /* instrument reference type (see above) */
    UBYTE data1, data2; /* depends on the "type" field */
    char name[60];      /* instrument name */
    } RefInstrument;

/* ---------- TRAK -----------------------------------------------------*/
```

iffp/smus.h Page 2

```
/* TRAK chunk contains an SEvent[]. */

/* SEvent: Simple musical event. */
typedef struct {
    UBYTE sID;           /* SEvent type code */
    UBYTE data;          /* sID-dependent data */
    } SEvent;

/* SEvent type codes "sID". */
#define SID_FirstNote     0
#define SID_LastNote    127     /* sIDs in the range SID_FirstNote through
                                 * SID_LastNote (sign bit = 0) are notes. The
                                 * sID is the MIDI tone number (pitch). */
#define SID_Rest        128     /* a rest; same data format as a note. */

#define SID_Instrument  129     /* set instrument number for this track. */
#define SID_TimeSig     130     /* set time signature for this track. */
#define SID_KeySig      131     /* set key signature for this track. */
#define SID_Dynamic     132     /* set volume for this track. */
#define SID_MIDI_Chnl   133     /* set MIDI channel number (sequencers) */
#define SID_MIDI_Preset 134     /* set MIDI preset number (sequencers) */
#define SID_Clef        135     /* inline clef change.
                                 * 0=Treble, 1=Bass, 2=Alto, 3=Tenor. */
#define SID_Tempo       136     /* Inline tempo change in beats per minute.*/

/* SID values 144 through 159: reserved for Instant Music SEvents. */

/* The remaining sID values up through 254: reserved for future
 * standardization. */
#define SID_Mark        255     /* SID reserved for an end-mark in RAM. */


/* ---------- SEvent FirstNote..LastNote or Rest -----------------------*/
typedef struct {
    unsigned tone     :8,       /* MIDI tone number 0 to 127; 128 = rest */
             chord    :1,       /* 1 = a chorded note */
             tieOut   :1,       /* 1 = tied to the next note or chord */
             nTuplet  :2,       /* 0 = none, 1 = triplet, 2 = quintuplet,
                                 * 3 = septuplet */
             dot      :1,       /* dotted note; multiply duration by 3/2 */
             division :3;       /* basic note duration is 2**-division:
                                 * 0 = whole note, 1 = half note, 2 = quarter
                                 * note, ... 7 = 128th note */
    } SNote;

/* Warning: An SNote is supposed to be a 16-bit entity.
 * Some C compilers will not pack bit fields into anything smaller
 * than an int. So avoid the actual use of this type unless you are certain
 * that the compiler packs it into a 16-bit word.
 */


/* You may get better object code by masking, ORing, and shifting using the
 * following definitions rather than the bit-packed fields, above. */
#define noteChord  (1<<7)       /* note is chorded to next note */

#define noteTieOut (1<<6)       /* note/chord is tied to next note/chord */

#define noteNShift 4                    /* shift count for nTuplet field */
#define noteN3     (1<<noteNShift)      /* note is a triplet */
#define noteN5     (2<<noteNShift)      /* note is a quintuplet */
#define noteN7     (3<<noteNShift)      /* note is a septuplet */
#define noteNMask  noteN7               /* bit mask for the nTuplet field */

#define noteDot    (1<<3)               /* note is dotted */
```

iffp/smus.h Page 3

```
#define noteDShift 0                        /* shift count for division field */
#define noteD1     (0<<noteDShift)          /* whole note division */
#define noteD2     (1<<noteDShift)          /* half note division */
#define noteD4     (2<<noteDShift)          /* quarter note division */
#define noteD8     (3<<noteDShift)          /* eighth note division */
#define noteD16    (4<<noteDShift)          /* sixteenth note division */
#define noteD32    (5<<noteDShift)          /* thirty-secondth note division */
#define noteD64    (6<<noteDShift)          /* sixty-fourth note division */
#define noteD128   (7<<noteDShift)          /* 1/128 note division */
#define noteDMask  noteD128                 /* bit mask for the division field */

#define noteDurMask 0x3F                    /* bit mask for all duration fields
                                             * division, nTuplet, dot */

/* Field access: */
#define IsChord(snote)   (((UWORD)snote) & noteChord)
#define IsTied(snote)    (((UWORD)snote) & noteTieOut)
#define NTuplet(snote)   ((((UWORD)snote) & noteNMask) >> noteNShift)
#define IsDot(snote)     (((UWORD)snote) & noteDot)
#define Division(snote)  ((((UWORD)snote) & noteDMask) >> noteDShift)

/* ---------- TimeSig SEvent -------------------------------------------*/
typedef struct {
    unsigned type     :8,       /* = SID_TimeSig */
             timeNSig :5,       /* time signature "numerator" timeNSig + 1 */
             timeDSig :3;       /* time signature "denominator" is
                                 * 2**timeDSig: 0 = whole note, 1 = half
                                 * note, 2 = quarter note, ...
                                 * 7 = 128th note */
    } STimeSig;

#define timeNMask  0xF8         /* bit mask for timeNSig field */
#define timeNShift 3            /* shift count for timeNSig field */

#define timeDMask  0x07         /*  bit mask for timeDSig field */

/* Field access: */
#define TimeNSig(sTime)  ((((UWORD)sTime) & timeNMask) >> timeNShift)
#define TimeDSig(sTime)   (((UWORD)sTime) & timeDMask)

/* ---------- KeySig SEvent -------------------------------------------*/
/* "data" value 0 = Cmaj; 1 through 7 = G,D,A,E,B,F#,C#;
 * 8 through 14 = F,Bb,Eb,Ab,Db,Gb,Cb.                                 */

/* ---------- Dynamic SEvent ------------------------------------------*/
/* "data" value is a MIDI key velocity 0..127. */


/* ---------- SMUS Writer Support Routines -----------------------------*/

/* Just call this to write a SHDR chunk. */
#define PutSHDR(iff, ssHdr)  \
    PutCk(iff, ID_SHDR, sizeof(SScoreHeader), (BYTE *)ssHdr)

#endif
```

```
/* 24bitDemo.c 05/91  C. Scheppner CBM
 *
 * Example which creates a 24-bit raster, saves it as a 24-bit ILBM,
 *   then loads it as a brush and shows it to you 4 planes at a time
 *   Optionally (if given a filename) just displays 4 planes at a time.
 *
 * requires linkage with several IFF modules
 * see Makefile
 */

#include "iffp/ilbmapp.h"


#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void cleanup(void);
void bye(UBYTE *s,int error);

#define MINARGS 1
char *vers = "\0$VER: 24bitDemo 37.5";
char *Copyright = "24bitDemo v37.5 (Freely Redistributable)";
char *usage = "Usage: 24bitDemo [loadname] (saves/loads if no loadname given)";


struct Library *IntuitionBase  = NULL;
struct Library *GfxBase        = NULL;
struct Library *IFFParseBase   = NULL;

/* Note - these fields are also available in the ILBMInfo structure */
struct   Screen         *scr;         /* for ptr to screen structure */
struct   Window         *win;         /* for ptr to window structure */
struct   RastPort       *wrp;         /* for ptr to RastPort  */
struct   ViewPort       *vp;          /* for ptr to Viewport  */


struct   NewWindow      mynw = {
   0, 0,                                 /* LeftEdge and TopEdge */
   0, 0,                                 /* Width and Height */
   -1, -1,                               /* DetailPen and BlockPen */
   VANILLAKEY|MOUSEBUTTONS,              /* IDCMP Flags with Flags below */
   BACKDROP|BORDERLESS|SMART_REFRESH|NOCAREREFRESH|ACTIVATE|RMBTRAP,
   NULL, NULL,                           /* Gadget and Image pointers */
   NULL,                                 /* Title string */
   NULL,                                 /* Screen ptr null till opened */
   NULL,                                 /* BitMap pointer */
   50, 20,                               /* MinWidth and MinHeight */
   0 , 0,                                /* MaxWidth and MaxHeight */
   CUSTOMSCREEN                          /* Type of window */
   };


BOOL    FromWb;


/* ILBM Property chunks to be grabbed
 * List BMHD, CMAP and CAMG first so we can skip them when we write
 * the file back out (they will be written out with separate code)
 */
LONG     ilbmprops[] = {
                ID_ILBM, ID_BMHD,
                ID_ILBM, ID_CMAP,
                ID_ILBM, ID_CAMG,
                ID_ILBM, ID_CCRT,
```

```
                ID_ILBM, ID_AUTH,
                ID_ILBM, ID_Copyright,
                TAG_DONE
                };

/* ILBM Collection chunks (more than one in file) to be gathered */
LONG     ilbmcollects[] = {
                ID_ILBM, ID_CRNG,
                TAG_DONE
                };

/* ILBM Chunk to stop on */
LONG     ilbmstops[] = {
                ID_ILBM, ID_BODY,
                TAG_DONE
                };


UBYTE nomem[]   = "Not enough memory\n";
UBYTE noiffh[]  = "Can't alloc iff\n";


/* For our allocated ILBM frames */
struct ILBMInfo   *ilbm[2];

#define SCRPLANES 4

USHORT colortable[32];
USHORT cstarts[]= { 0x000, 0x800, 0x000, 0x080, 0x000, 0x008 };
USHORT coffs[]  = { 0x100, 0x100, 0x010, 0x010, 0x001, 0x001 };

UBYTE *ilbmname = "RAM:24bit.ilbm";
UBYTE *rgbnames[]={"R0","R1","R2","R3","R4","R5","R6","R7",
                   "G0","G1","G2","G3","G4","G5","G6","G7",
                   "B0","B1","B2","B3","B4","B5","B6","B7" };

UBYTE *endtext1 = "Displayed 24 planes, 4 at a time.";
UBYTE *endtext2 = "Press mousebutton or key to exit.";

/*
 * MAIN
 */
void main(int argc, char **argv)
   {
    struct RastPort *rp = NULL;
    struct BitMap dummy = {0};
    struct BitMap *bm = NULL, *xbm, *sbm;
    LONG          error = 0L;
    USHORT        width, height, depth, pwidth, pheight, pmode, extra, rgb;
    ULONG         plsize;
    UBYTE         *tpp;
    BOOL          DoSave = TRUE;
    int           k, p, s, n;

   FromWb = argc ? FALSE : TRUE;

   if((argc > 1)&&(argv[argc-1][0]=='?'))
        {
        printf("%s\n%s\n",Copyright,usage);
        bye("",RETURN_OK);
        }

   if(argc==2)
        {
        ilbmname = argv[1];
        DoSave = FALSE;
        }

    /* Open Libraries */

    if(!(IntuitionBase = OpenLibrary("intuition.library", 0)))
       bye("Can't open intuition library.\n",RETURN_WARN);

    if(!(GfxBase = OpenLibrary("graphics.library",0)))
       bye("Can't open graphics library.\n",RETURN_WARN);

    if(!(IFFParseBase = OpenLibrary("iffparse.library",0)))
       bye("Can't open iffparse library.\n",RETURN_WARN);


/*
 * Alloc ILBMInfo structs
 */
     if(!(ilbm[0] = (struct ILBMInfo *)
         AllocMem(sizeof(struct ILBMInfo),MEMF_PUBLIC|MEMF_CLEAR)))
                bye(nomem,RETURN_FAIL);
     if(!(ilbm[1] = (struct ILBMInfo *)
         AllocMem(sizeof(struct ILBMInfo),MEMF_PUBLIC|MEMF_CLEAR)))
                bye(nomem,RETURN_FAIL);

/*
 * Here we set up our ILBMInfo fields for our
 * application.
 * Above we have defined the propery and collection chunks
 * we are interested in (some required like BMHD)
 */

     ilbm[0]->ParseInfo.propchks        = ilbmprops;
     ilbm[0]->ParseInfo.collectchks     = ilbmcollects;
     ilbm[0]->ParseInfo.stopchks        = ilbmstops;

     ilbm[0]->windef     = &mynw;

     *ilbm[1] = *ilbm[0];


/*
 * Alloc IFF handles for frame
 */
     if(!(ilbm[0]->ParseInfo.iff = AllocIFF())) bye(noiffh,RETURN_FAIL);
     if(!(ilbm[1]->ParseInfo.iff = AllocIFF())) bye(noiffh,RETURN_FAIL);

/* for saving our demo 24-bit ILBM */

     width  = 320;
     height = 200;
     depth  = 24;

     /* Page width, height, and mode for saved ILBM */
     pwidth  = width  < 320 ? 320 : width;
     pheight = height < 200 ? 200 : height;
     pmode   = pwidth >= 640  ? HIRES : 0L;
     pmode  |= pheight >= 400 ? LACE  : 0L;

     plsize = RASSIZE(width,height);

     if(!DoSave) goto nosave;

     /*
      * Allocate Bitmap and planes
      */
```

```
     extra = depth > 8 ? depth - 8 : 0;
     if(ilbm[0]->brbitmap = AllocMem(sizeof(struct BitMap) + (extra<<2),
                                          MEMF_CLEAR))
        {
        bm = ilbm[0]->brbitmap;
        InitBitMap(bm,depth,width,height);
        for(k=0, error=0; k<depth && (!error); k++)
            {
            if(!(bm->Planes[k] = AllocRaster(width,height)))
                        error = IFFERR_NOMEM;
            if(! error)
                {
                BltClear(bm->Planes[k], RASSIZE(width,height),0);
                }
            }

        if(!error)
            {
            if(!(rp = AllocMem(sizeof(struct RastPort),MEMF_CLEAR)))
                error = IFFERR_NOMEM;
            else
                {
                InitRastPort(rp);
                rp->BitMap = bm;
                rp->Mask = 0x01;         /* we'll render 1 plane at a time */
                SetAPen(rp,1);
                SetDrMd(rp,JAM1);
                }
            }

        if(!error)
            {
            /* Put something recognizable in the planes.
             * Our bitmap is not part of a screen or viewport
             * so we can fiddle with the pointers and depth
             */
            tpp = bm->Planes[0];         /* save first plane pointer */
            bm->Depth = 1;
            for(k=0; k<depth; k++)       /* swap in planeptrs 1 at a time */
                {
                bm->Planes[0] = bm->Planes[k];
                Move(rp,k * 10, (k * 8) + 8);    /* render rgb bitname text */
                Text(rp, rgbnames[k], 2);
                }
            bm->Depth = depth;           /* restore depth */
            bm->Planes[0] = tpp;         /* and first pointer */

            /* Save the 24-bit ILBM */
            printf("Saving %s\n",ilbmname);
            error = saveilbm(ilbm[0], ilbm[0]->brbitmap, pmode,
                width,  height, pwidth, pheight,
                NULL, 0, 0,     /* colortable */
                mskNone, 0,     /* masking, transparent */
                NULL, NULL,     /* chunklists */
                ilbmname);
            }

        /* Free our bitmap */
        for(k=0; k<depth; k++)
            {
            if(ilbm[0]->brbitmap->Planes[k])
                    FreeRaster(ilbm[0]->brbitmap->Planes[k],width,height);
            }
        FreeMem(ilbm[0]->brbitmap, sizeof(struct BitMap) + (extra << 2));
        ilbm[0]->brbitmap = NULL;
        if(rp)  FreeMem(rp, sizeof(struct RastPort));
        }

    if(error)
        {
        printf("%s\n",IFFerr(error));
        bye(" ", RETURN_FAIL);
        }

nosave:

/* Normally you would use showilbm() to open an appropriate acreen
 * and display an ILBM in it.  However, this is a 24-bit ILBM
 * so we will load it as a brush (bitmap).
 * Here we are demonstrating
 *  - first querying an ILBM to get its BMHD and CAMG (real or computed)
 *  - then opening our own display
 *  - then loading the 24-bit ILBM as a brush (bitmap) and displaying
 *      it 4 planes at a time in our 4-plane screen.
 */

    printf("Attempting to load %s as a bitmap and display 4 planes at a time\n",
                ilbmname);

    if(!(error = queryilbm(ilbm[0],ilbmname)))
        {
        D(bug("24bitDemo: after query, this ILBM is %ld x %ld x %ld,modeid=$%lx\n",
          ilbm[0]->Bmhd.w, ilbm[0]->Bmhd.h, ilbm[0]->Bmhd.nPlanes, ilbm[0]->camg));

        /* Note - you could use your own routines to open your
         * display, but if so, you must initialize ilbm[0]->scr,
         * ilbm[0]->win, ilbm[0]->wrp, ilbm[0]->srp, and ilbm[0]->vp for your
         * display.  Here we will use opendisplay() which will initialize
         * those fields.
         */

        if(!(opendisplay(ilbm[0],
                        MAX(ilbm[0]->Bmhd.pageWidth, ilbm[0]->Bmhd.w),
                        MAX(ilbm[0]->Bmhd.pageHeight,ilbm[0]->Bmhd.h),
                        MIN(ilbm[0]->Bmhd.nPlanes, SCRPLANES),
                        ilbm[0]->camg)))
            {
            printf("Failed to open display\n");
            }
        else
            {
            D(bug("24bitDemo: opendisplay (%ld planes) successful\n",SCRPLANES));

            scr = ilbm[0]->scr;
            win = ilbm[0]->win;
            wrp = ilbm[0]->wrp;
            vp  = ilbm[0]->vp;

            if(!(error = loadbrush(ilbm[1], ilbmname)))
                {
                D(bug("24bitDemo: loadbrush successful\n"));

                /* Note - we don't need to examine or copy any
                 * chunks from the file, so we will close file now
                 */
                closeifile(ilbm[0]);
                ScreenToFront(ilbm[0]->scr);

                xbm = &dummy;           /* spare bitmap */
                sbm = &scr->BitMap;     /* screen's bitmap */
                bm = ilbm[1]->brbitmap; /* the 24-plane bitmap */
                depth = bm->Depth;
```

```
                InitBitMap(xbm,SCRPLANES,scr->Width,scr->Height);

                /* Show the 24 planes */
                for(p=0; p<depth; p+=SCRPLANES) /* 4 at a time */
                    {
                    SetRast(&scr->RastPort, 0);
                    for(s=0; s<SCRPLANES; s++)
                        {
                        if((p+s) < depth) xbm->Planes[s] = bm->Planes[p+s];
                        else              xbm->Planes[s] = NULL, xbm->Depth--;
                        }
                    /* Blit planes to the screen */
                    BltBitMap(xbm, 0, 0,
                              sbm, 0, 0,
                              scr->Width, scr->Height,
                              0xC0, 0x0F, NULL);

                    /* Emulate 8-bit color with 4-bit per gun colors
                     * by using each rgb value twice
                     */
                    for(n=0, rgb=cstarts[p /SCRPLANES]; n < 16; n++)
                        {
                        if(!n)  colortable[n] = 0xFFF;
                        else    colortable[n] = rgb;
                        /* bump gun for every 2 planes since
                         * we only have 8 bits per gun
                         */
                        if(n & 1)  rgb += coffs[ p / SCRPLANES];
                        }
                    LoadRGB4(vp, colortable, 16);
                    Delay(50);
                    }

                SetRast(&scr->RastPort, 0);

                SetAPen(wrp, 1);
                Move(wrp, 24, 80);
                Text(wrp, endtext1, strlen(endtext1));
                Move(wrp, 24, 120);
                Text(wrp, endtext2, strlen(endtext2));

                Wait(1<<win->UserPort->mp_SigBit);
                unloadbrush(ilbm[1]);   /* deallocs colors, closeifile if needed */
                }
            closedisplay(ilbm[0]);
            printf("Done\n");
            }
        }

    if(error)   printf("%s\n",IFFerr(error));

    cleanup();
    exit(RETURN_OK);
    }


void bye(UBYTE *s,int error)
    {
    if((*s)&&(!FromWb)) printf("%s\n",s);
    cleanup();
    exit(error);
    }
```

```
void cleanup()
   {
   if(ilbm[0])
        {
        if(ilbm[0]->ParseInfo.iff)      FreeIFF(ilbm[0]->ParseInfo.iff);
        FreeMem(ilbm[0],sizeof(struct ILBMInfo));
        }
   if(ilbm[1])
        {
        if(ilbm[1]->ParseInfo.iff)      FreeIFF(ilbm[1]->ParseInfo.iff);
        FreeMem(ilbm[1],sizeof(struct ILBMInfo));
        }

   if(GfxBase)          CloseLibrary(GfxBase);
   if(IntuitionBase)    CloseLibrary(IntuitionBase);
   if(IFFParseBase)     CloseLibrary(IFFParseBase);
   }
```

```
/* ILBMDemo.c  05/91   C. Scheppner CBM
 *
 * Demonstrates displaying an ILBM, loading a brush,
 *   saving an ILBM, and optionally printing a screen (CTRL-p)
 *   Use -c (or -c1, -c2, etc) as filename to read from or save to clipboard.
 *
 * requires linkage with several iffp modules - see Makefile
 */

#include "iffp/ilbmapp.h"


#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void chkmsg(void);
void cleanup(void);
void bye(UBYTE *s,int error);

#define SAVECHANGES

#define MINARGS 3
char *vers = "\0$VER: ILBMDemo 37.5";
char *Copyright = "ILBMDemo v37.5 (Freely Redistributable)";
char *usage =
"Usage: ILBMDemo sourceilbm destilbm [brushname]  (CTRL-p to print screen)\n"
"Displays source, optionally loads and blits brush, saves to dest\n"
"Use filename -c[unit] (ie. -c, -c1, -c2, etc.) for clipboard\n";

char *savename;

struct Library *IntuitionBase  = NULL;
struct Library *GfxBase        = NULL;
struct Library *IFFParseBase   = NULL;

/* Note - these fields are also available in the ILBMInfo structure */
struct   Screen         *scr;         /* for ptr to screen structure */
struct   Window         *win;         /* for ptr to window structure */
struct   RastPort       *wrp;         /* for ptr to RastPort  */
struct   ViewPort       *vp;          /* for ptr to Viewport  */

struct   IntuiMessage   *msg;

struct   NewWindow      mynw = {
   0, 0,                                  /* LeftEdge and TopEdge */
   0, 0,                                  /* Width and Height */
   -1, -1,                                /* DetailPen and BlockPen */
   VANILLAKEY|MOUSEBUTTONS,               /* IDCMP Flags with Flags below */
   BACKDROP|BORDERLESS|SMART_REFRESH|NOCAREREFRESH|ACTIVATE|RMBTRAP,
   NULL, NULL,                            /* Gadget and Image pointers */
   NULL,                                  /* Title string */
   NULL,                                  /* Screen ptr null till opened */
   NULL,                                  /* BitMap pointer */
   50, 20,                                /* MinWidth and MinHeight */
   0 , 0,                                 /* MaxWidth and MaxHeight */
   CUSTOMSCREEN                           /* Type of window */
   };


BOOL   FromWb, Done;


/* ILBM Property chunks to be grabbed
 * List BMHD, CMAP and CAMG first so we can skip them when we write
 * the file back out (they will be written out with separate code)
 */
LONG    ilbmprops[] = {
                ID_ILBM, ID_BMHD,
                ID_ILBM, ID_CMAP,
                ID_ILBM, ID_CAMG,
                ID_ILBM, ID_CCRT,
                ID_ILBM, ID_AUTH,
                ID_ILBM, ID_Copyright,
                TAG_DONE
                };

/* ILBM Collection chunks (more than one in file) to be gathered */
LONG    ilbmcollects[] = {
                ID_ILBM, ID_CRNG,
                TAG_DONE
                };

/* ILBM Chunk to stop on */
LONG    ilbmstops[] = {
                ID_ILBM, ID_BODY,
                TAG_DONE
                };


/* For test of adding new chunks to saved FORM */
struct Chunk newchunks[2] = {
        {
        &newchunks[1],
        ID_ILBM, ID_AUTH, IFFSIZE_UNKNOWN,
        "CAS_CBM"},
        {
        NULL,
        ID_ILBM, ID_NAME, IFFSIZE_UNKNOWN,
        "Untitled No. 27"},
        };


UBYTE nomem[]  = "Not enough memory\n";
UBYTE noiffh[] = "Can't alloc iff\n";

/* our indexes to reference our frames
 * DEFault, BRUsh, and SCReen
 */
#define DEF     0
#define BRU     1
#define SCR     2
#define UICOUNT 3

/* For our ILBM frames */
struct ILBMInfo  *ilbms[UICOUNT]  = { NULL };


/*
 * MAIN
 */
void main(int argc, char **argv)
   {
#ifdef SAVECHANGES
   struct Chunk *chunk;
   CamgChunk *camg;
   LONG saverror;
#endif
   UBYTE *ilbmname=NULL, *brushname=NULL, ans, c;
   BPTR lock;
   LONG error;
```

```
   FromWb = argc ? FALSE : TRUE;

   if((argc<MINARGS)||(argv[argc-1][0]=='?'))
        {
        printf("%s\n%s\n",Copyright,usage);
        bye("",RETURN_OK);
        }

   switch(argc)
      {
      case 4:
         brushname      = argv[3];
      case 3:
         savename       = argv[2];
         ilbmname       = argv[1];
         break;
      }

   /* if dest not clipboard, warn if dest file already exists */
   if(strcmp(savename,"-c"))
        {
        if(lock = Lock(savename,ACCESS_READ))
            {
            UnLock(lock);
            printf("Dest file \"%s\" already exists.  Overwrite (y or n) ? ",
                        savename);
            ans = 0;
            while((c = getchar()) != '\n') if(!ans)  ans = c | 0x20;
            if(ans == 'n')   bye("Exiting.\n",RETURN_OK);
            }
        }

   /* Open Libraries */

   if(!(IntuitionBase = OpenLibrary("intuition.library", 0)))
      bye("Can't open intuition library.\n",RETURN_WARN);

   if(!(GfxBase = OpenLibrary("graphics.library",0)))
      bye("Can't open graphics library.\n",RETURN_WARN);

   if(!(IFFParseBase = OpenLibrary("iffparse.library",0)))
      bye("Can't open iffparse library.\n",RETURN_WARN);


/*
 * Alloc three ILBMInfo structs (one each for defaults, screen, brush)
 */
    if(!(ilbms[0] = (struct ILBMInfo *)
        AllocMem(UICOUNT * sizeof(struct ILBMInfo),MEMF_PUBLIC|MEMF_CLEAR)))
                bye(nomem,RETURN_FAIL);
    else
        {
        ilbms[BRU] = ilbms[0] + 1;
        ilbms[SCR] = ilbms[0] + 2;
        }

/*
 * Here we set up default ILBMInfo fields for our
 * application's frames.
 * Above we have defined the propery and collection chunks
 * we are interested in (some required like BMHD)
 * Since all of our frames are for ILBM's, we'll initialize
 * one default frame and clone the others from it.
 */
```

```
    ilbms[DEF]->ParseInfo.propchks      = ilbmprops;
    ilbms[DEF]->ParseInfo.collectchks   = ilbmcollects;
    ilbms[DEF]->ParseInfo.stopchks      = ilbmstops;

    ilbms[DEF]->windef  = &mynw;
/*
 * Initialize our working ILBM frames from our default one
 */
    *ilbms[SCR] = *ilbms[DEF];  /* for our screen */
    *ilbms[BRU] = *ilbms[DEF];  /* for our brush  */

/*
 * Alloc two IFF handles (one for screen frame, one for brush frame)
 */
    if(!(ilbms[SCR]->ParseInfo.iff = AllocIFF())) bye(noiffh,RETURN_FAIL);
    if(!(ilbms[BRU]->ParseInfo.iff = AllocIFF())) bye(noiffh,RETURN_FAIL);

/* Load and display an ILBM
 */
    if(error = showilbm(ilbms[SCR],ilbmname))
        {
        printf("Can't load background \"%s\"\n",ilbmname);
        bye("",RETURN_WARN);
        }

    /* These were set up by our successful showilbm() above */
    win = ilbms[SCR]->win;      /* our window */
    wrp = ilbms[SCR]->wrp;      /* our window's RastPort */
    scr = ilbms[SCR]->scr;      /* our screen */
    vp  = ilbms[SCR]->vp;              /* our screen's ViewPort */

    ScreenToFront(scr);


 /* Now let's load a brush and blit it into the window
  */
    if(brushname)
        {
        if (error = loadbrush(ilbms[BRU],brushname))
            {
            printf("Can't load brush \"%s\"\n",brushname);
            bye("",RETURN_WARN);
            }
        else    /* Success */
            {
            D(bug("About to Blt bitmap $%lx to rp $%lx, w=%ld h=%ld\n",
                ilbms[BRU]->brbitmap,wrp,ilbms[BRU]->Bmhd.w,ilbms[BRU]->Bmhd.h));
            BltBitMapRastPort(ilbms[BRU]->brbitmap,0,0,
                                wrp,0,0,
                                ilbms[BRU]->Bmhd.w, ilbms[BRU]->Bmhd.h,
                                0xC0);
            }
        }

#ifdef SAVECHANGES

 /* This code is an example for Read/Modify/Write programs
  *
  * We copy off the parsed chunks we want to preserve,
  * close the IFF read file, reopen it for write,
  * and save a new ILBM which
  * will include the chunks we have preserved, but
  * with newly computed and set-up BMHD, CMAP, and CAMG.
  */

   if(!(ilbms[SCR]->ParseInfo.copiedchunks =
```

```
        copychunks(ilbms[SCR]->ParseInfo.iff,
                   ilbmprops, ilbmcollects,
                   MEMF_PUBLIC)))
                printf("error cloning chunks\n");
    else
        {
        /* we can close the file now */
        closeifile(ilbms[SCR]);

        printf("Test of copychunks and findchunk:\n");

        /* Find copied CAMG chunk if any */
        if(chunk = findchunk(ilbms[SCR]->ParseInfo.copiedchunks,ID_ILBM,ID_CAMG))
            {
            camg = (CamgChunk *)chunk->ch_Data;
            printf("CAMG: $%08lx\n",camg->ViewModes);
            }
        else printf("No CAMG found\n");

        /* Find copied CRNG chunks if any */
        if(chunk = findchunk(ilbms[SCR]->ParseInfo.copiedchunks,ID_ILBM,ID_CRNG))
            {
            while((chunk)&&(chunk->ch_ID == ID_CRNG))
                {
                printf("Found a CRNG chunk\n");
                chunk = chunk->ch_Next;
                }
            }
        else printf("No CRNG chunks found\n");
        }

    printf("\nAbout to save screen as %s, adding NAME and AUTH chunks\n",
                savename);

    if(saverror = screensave(ilbms[SCR], ilbms[SCR]->scr,
                             ilbms[SCR]->ParseInfo.copiedchunks,
                             newchunks,
                             savename))
                  printf("%s\n",IFFerr(saverror));

#endif

    Done = FALSE;
    while(!Done)
       {
       Wait(1<<win->UserPort->mp_SigBit);
       chkmsg();
       }


    cleanup();
    exit(RETURN_OK);
    }
```

```
void chkmsg(void)
    {
    LONG  error;
    ULONG class;
    UWORD code;
    WORD  mousex, mousey;

    while(msg = (struct IntuiMessage *)GetMsg(win->UserPort))
        {
        class = msg->Class;
        code  = msg->Code;
        mousex = msg->MouseX;
        mousey = msg->MouseY;

        ReplyMsg(msg);
        switch(class)
            {
            case MOUSEBUTTONS:
            switch(code)
                {
                /* emulate a close gadget */
                case SELECTDOWN:
                   if((mousex < 12)&&(mousey < 12))     Done = TRUE;
                   break;
                default:
                   break;
                }
            case VANILLAKEY:
            switch(code)
                {
                /* also quit on CTRL-C, CTRL-D, or q */
                case 'q': case 0x04: case 0x03:
                  Done = TRUE;
                  break;
                case 0x10:      /* CTRL-p means print */

                  /* Print the whole screen */
                  if(error=screendump(ilbms[SCR]->scr,
                                0,0,
                                ilbms[SCR]->scr->Width,
                                ilbms[SCR]->scr->Height,
                                0,0))
                        printf("Screendump printer error=%ld\n",error);
                  break;

                default:
                  break;
                }
            default:
            break;
            }
        }
    }


void bye(UBYTE *s,int error)
   {
   if((*s)&&(!FromWb)) printf("%s\n",s);
   cleanup();
   exit(error);
   }
```

```
void cleanup()
   {
   if(ilbms[SCR])
        {
        if(ilbms[SCR]->scr)               unshowilbm(ilbms[SCR]);
#ifdef SAVECHANGES
        freechunklist(ilbms[SCR]->ParseInfo.copiedchunks);
#endif
        if(ilbms[SCR]->ParseInfo.iff)    FreeIFF(ilbms[SCR]->ParseInfo.iff);
        }

   if(ilbms[BRU])
        {
        if(ilbms[BRU]->brbitmap)         unloadbrush(ilbms[BRU]);
        if(ilbms[BRU]->ParseInfo.iff)    FreeIFF(ilbms[BRU]->ParseInfo.iff);
        }

   if(ilbms[0])
        {
        FreeMem(ilbms[0],UICOUNT * sizeof(struct ILBMInfo));
        }

   if(GfxBase)       CloseLibrary(GfxBase);
   if(IntuitionBase) CloseLibrary(IntuitionBase);
   if(IFFParseBase)  CloseLibrary(IFFParseBase);
   }
```

```
/* ILBMLoad.c 05/91  C. Scheppner CBM
 *
 * Example which
 *   - first queries an ILBM to determine size and mode
 *   - then opens an appropriate screen and window
 *   - then loads the ILBM into the already opened screen
 *
 * For clipboard, use filename -c[unit] (like -c, -c1, -c2, etc.)
 *
 * requires linkage with several IFF modules
 * see Makefile
 */

#include "iffp/ilbmapp.h"


#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void cleanup(void);
void bye(UBYTE *s,int error);

#define MINARGS 2
char *vers = "\0$VER: ILBMLoad 37.5";
char *Copyright = "ILBMLoad v37.5 (Freely Redistributable)";
char *usage = "Usage: ILBMLoad ilbmname (-c[unit] for clipboard";


struct Library *IntuitionBase  = NULL;
struct Library *GfxBase        = NULL;
struct Library *IFFParseBase   = NULL;

/* Note - these fields are also available in the ILBMInfo structure */
struct   Screen         *scr;         /* for ptr to screen structure */
struct   Window         *win;         /* for ptr to window structure */
struct   RastPort       *wrp;         /* for ptr to RastPort  */
struct   ViewPort       *vp;          /* for ptr to Viewport  */

struct   IntuiMessage   *msg;

struct   NewWindow      mynw = {
   0, 0,                                 /* LeftEdge and TopEdge */
   0, 0,                                 /* Width and Height */
   -1, -1,                               /* DetailPen and BlockPen */
   VANILLAKEY|MOUSEBUTTONS,              /* IDCMP Flags with Flags below */
   BACKDROP|BORDERLESS|SMART_REFRESH|NOCAREREFRESH|ACTIVATE|RMBTRAP,
   NULL, NULL,                           /* Gadget and Image pointers */
   NULL,                                 /* Title string */
   NULL,                                 /* Screen ptr null till opened */
   NULL,                                 /* BitMap pointer */
   50, 20,                               /* MinWidth and MinHeight */
   0 , 0,                                /* MaxWidth and MaxHeight */
   CUSTOMSCREEN                          /* Type of window */
   };


BOOL    FromWb;


/* ILBM Property chunks to be grabbed
 * List BMHD, CMAP and CAMG first so we can skip them when we write
 * the file back out (they will be written out with separate code)
 */
LONG    ilbmprops[] = {
```

```
                ID_ILBM, ID_BMHD,
                ID_ILBM, ID_CMAP,
                ID_ILBM, ID_CAMG,
                ID_ILBM, ID_CCRT,
                ID_ILBM, ID_AUTH,
                ID_ILBM, ID_Copyright,
                TAG_DONE
                };

/* ILBM Collection chunks (more than one in file) to be gathered */
LONG    ilbmcollects[] = {
                ID_ILBM, ID_CRNG,
                TAG_DONE
                };

/* ILBM Chunk to stop on */
LONG    ilbmstops[] = {
                ID_ILBM, ID_BODY,
                TAG_DONE
                };


UBYTE nomem[]   = "Not enough memory\n";
UBYTE noiffh[] = "Can't alloc iff\n";



/* For our allocated ILBM frame */
struct ILBMInfo  *ilbm;


/*
 * MAIN
 */
void main(int argc, char **argv)
   {
   UBYTE *ilbmname=NULL;
   LONG error = 0L;

   FromWb = argc ? FALSE : TRUE;

   if((argc<MINARGS)||(argv[argc-1][0]=='?'))
        {
        printf("%s\n%s\n",Copyright,usage);
        bye("",RETURN_OK);
        }

   ilbmname = argv[1];

   /* Open Libraries */

   if(!(IntuitionBase = OpenLibrary("intuition.library", 0)))
      bye("Can't open intuition library.\n",RETURN_WARN);

   if(!(GfxBase = OpenLibrary("graphics.library",0)))
      bye("Can't open graphics library.\n",RETURN_WARN);

   if(!(IFFParseBase = OpenLibrary("iffparse.library",0)))
      bye("Can't open iffparse library.\n",RETURN_WARN);



/*
 * Alloc one ILBMInfo struct
 */
    if(!(ilbm = (struct ILBMInfo *)
        AllocMem(sizeof(struct ILBMInfo),MEMF_PUBLIC|MEMF_CLEAR)))
                bye(nomem,RETURN_FAIL);

/*
 * Here we set up our ILBMInfo fields for our
 * application.
 * Above we have defined the propery and collection chunks
 * we are interested in (some required like BMHD)
 */

    ilbm->ParseInfo.propchks    = ilbmprops;
    ilbm->ParseInfo.collectchks = ilbmcollects;
    ilbm->ParseInfo.stopchks    = ilbmstops;

    ilbm->windef        = &mynw;

/*
 * Alloc IFF handle for frame
 */
    if(!(ilbm->ParseInfo.iff = AllocIFF())) bye(noiffh,RETURN_FAIL);

/* Normally you would use showilbm() to open an appropriate acreen
 * and display an ILBM in it.
 *
 * However, here we are demonstrating
 *  - first querying an ILBM to get its BMHD and CAMG (real or computed)
 *  - then opening our own display
 *  - then loading the ILBM into it
 */

    if(!(error = queryilbm(ilbm,ilbmname)))
        {
        D(bug("ilbmload: after query, this ILBM is %ld x %ld x %ld, modeid=$%lx\n",
                ilbm->Bmhd.w, ilbm->Bmhd.h, ilbm->Bmhd.nPlanes, ilbm->camg));

        /* Note - you could use your own routines to open your
         * display, but if so, you must initialize ilbm->scr,
         * ilbm->win, ilbm->wrp, ilbm->srp, and ilbm->vp for your display.
         * Here we will use opendisplay() which will initialize
         * those fields.
         */
        if(!(opendisplay(ilbm,
                        MAX(ilbm->Bmhd.pageWidth, ilbm->Bmhd.w),
                        MAX(ilbm->Bmhd.pageHeight,ilbm->Bmhd.h),
                        MIN(ilbm->Bmhd.nPlanes,MAXAMDEPTH),
                        ilbm->camg)))
            {
            printf("Failed to open display\n");
            }
        else
            {
            D(bug("ilbmload: opendisplay successful\n"));

            scr = ilbm->scr;
            win = ilbm->win;

            if(!(error = loadilbm(ilbm, ilbmname)))
                {
                D(bug("ilbmload: loadilbm successful\n"));

                /* Note - we don't need to examine or copy any
                 * chunks from the file, so we will close file now
                 */
                closeifile(ilbm);
                ScreenToFront(ilbm->scr);
                Wait(1<<win->UserPort->mp_SigBit);
```

```
                unloadilbm(ilbm);       /* deallocs colors, closeifile if needed */
                }
            closedisplay(ilbm);
            }
        }

    if(error)   printf("%s\n",IFFerr(error));

    cleanup();
    exit(RETURN_OK);
    }


void bye(UBYTE *s,int error)
   {
   if((*s)&&(!FromWb)) printf("%s\n",s);
   cleanup();
   exit(error);
   }


void cleanup()
   {
   if(ilbm)
        {
        if(ilbm->ParseInfo.iff)         FreeIFF(ilbm->ParseInfo.iff);
        FreeMem(ilbm,sizeof(struct ILBMInfo));
        }

   if(GfxBase)          CloseLibrary(GfxBase);
   if(IntuitionBase)    CloseLibrary(IntuitionBase);
   if(IFFParseBase)     CloseLibrary(IFFParseBase);
   }
```

apps/ILBMtoC/ILBMtoC.c Page 1

```
/*----------------------------------------------------------------*/
/*                                                                */
/* ILBMtoC: reads in ILBM, prints out ascii representation,       */
/*  for including in C files.                                     */
/*                                                                */
/* Based on ILBMDump.c by Jerry Morrison and Steve Shaw,          */
/* Electronic Arts.                                               */
/* Jan 31, 1986                                                   */
/*                                                                */
/* This software is in the public domain.                         */
/* This version for the Commodore-Amiga computer.                 */
/*                                                                */
/*  Callable from CLI ONLY                                        */
/*  modified 05-91 for use wuth iffparse modules                  */
/*  Requires linkage with several other modules - see Makefile   */
/*----------------------------------------------------------------*/

#include "iffp/ilbmapp.h"

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

char *vers = "\0$VER: ILBMtoC 37.5";
char *Copyright = "ILBMtoC v37.5 (Freely Redistributable)";

void GetSuffix(UBYTE *to, UBYTE *fr);
void bye(UBYTE *s, int e);
void cleanup(void);

struct Library *IFFParseBase = NULL;
struct Library *GfxBase = NULL;

/* ILBM frame */
struct ILBMInfo ilbm = {0};


/* ILBM Property chunks to be grabbed - only BMHD needed for this app
 */
LONG    ilbmprops[] = {
                ID_ILBM, ID_BMHD,
                TAG_DONE
                };

/* ILBM Collection chunks (more than one in file) to be gathered */
LONG    *ilbmcollects = NULL;   /* none needed for this app */

/* ILBM Chunk to stop on */
LONG    ilbmstops[] = {
                ID_ILBM, ID_BODY,
                TAG_DONE
                };


UBYTE defSwitch[] = "b";

/** main() ******************************************************************/

void main(int argc, char **argv)
    {
    UBYTE *sw;
    FILE *fp;
    LONG error=NULL;
    UBYTE *ilbmname, name[80], fname[80];
```

apps/ILBMtoC/ILBMtoC.c Page 2

```
    if ((argc < 2)||(argv[argc-1][0]=='?'))
        {
        printf("Usage from CLI: 'ILBMtoC filename switch-string'\n");
        printf(" where switch-string = \n");
        printf("  <nothing> : Bob format (default)\n");
        printf("  s         : Sprite format (with header and trailer words)\n");
        printf("  sn        : Sprite format (No header and trailer words)\n");
        printf("  a         : Attached sprite (with header and trailer)\n");
        printf("  an        : Attached sprite (No header and trailer)\n");
        printf(" Add 'c' to switch list to output CR's with LF's   \n");
        exit(RETURN_OK);
        }


    if(!(GfxBase = OpenLibrary("graphics.library",0)))
        bye("Can't open graphics.library",RETURN_FAIL);

    if(!(IFFParseBase = OpenLibrary("iffparse.library",0)))
        bye("Can't open iffparse.library",RETURN_FAIL);

/*
 * Here we set up default ILBMInfo fields for our
 * application's frames.
 * Above we have defined the propery and collection chunks
 * we are interested in (some required like BMHD)
 */
    ilbm.ParseInfo.propchks      = ilbmprops;
    ilbm.ParseInfo.collectchks   = ilbmcollects;
    ilbm.ParseInfo.stopchks      = ilbmstops;
    if(!(ilbm.ParseInfo.iff = AllocIFF()))
        bye(IFFerr(IFFERR_NOMEM),RETURN_FAIL);  /* Alloc an IFFHandle */

    sw = (argc>2) ? (UBYTE *)argv[2] : defSwitch;
    ilbmname = argv[1];

    if (error = loadbrush(&ilbm,ilbmname))
        {
        printf("Can't load ilbm \"%s\", ifferr=%s\n",ilbmname,IFFerr(error));
        bye("",RETURN_WARN);
        }
    else /* Successfully loaded ILBM */
        {
        printf(" Creating file %s.c \n",argv[1]);
        GetSuffix(name,argv[1]);
        strcpy(fname,argv[1]);
        strcat(fname,".c");
        fp = fopen(fname,"w");
        if(fp)
            {
            BMPrintCRep(ilbm.brbitmap,fp,name,sw);
            fclose(fp);
            }
        else  printf("Couldn't open output file: %s. \n", fname);
        unloadbrush(&ilbm);
        }
    printf("\n");
    bye("",RETURN_OK);
    }



/* this copies part of string after the last '/' or ':' */
void GetSuffix(to, fr) UBYTE *to, *fr; {
    int i;
    UBYTE c,*s = fr;
    for (i=0; ;i++) {
        c = *s++;
        if (c == 0) break;
        if (c == '/') fr = s;
        else if (c == ':') fr = s;
        }
    strcpy(to,fr);
    }


void bye(UBYTE *s, int e)
    {
    if(s&&(*s)) printf("%s\n",s);
    cleanup();
    exit(e);
    }


void cleanup()
    {
    if(ilbm.ParseInfo.iff)              FreeIFF(ilbm.ParseInfo.iff);

    if(IFFParseBase)    CloseLibrary(IFFParseBase);
    if(GfxBase)         CloseLibrary(GfxBase);
    }
```

```
/*----------------------------------------------------------------------*/
/*                                                                      */
/* ILBMtoRaw: reads in ILBM, writes out raw file (raw planes,           */
/*  followed by colormap)                                               */
/*                                                                      */
/* Based on ILBMRaw.c by Jerry Morrison and Steve Shaw,                 */
/* Electronic Arts.                                                     */
/* Jan 31, 1986                                                         */
/*                                                                      */
/* This software is in the public domain.                               */
/* This version for the Commodore-Amiga computer.                       */
/*                                                                      */
/*  Callable from CLI ONLY                                              */
/*  modified 05-91 for use wuth iffparse modules                        */
/*  Requires linkage with several other modules - see Makefile          */
/*----------------------------------------------------------------------*/

#include "iffp/ilbmapp.h"

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

char *vers = "\0$VER: ILBMtoRaw 37.5";
char *Copyright = "ILBMtoRaw v37.5 (Freely Redistributable)";

void bye(UBYTE *s, int e);
void cleanup(void);

LONG SaveBitMap(UBYTE *name, struct BitMap *bm, SHORT *cols, int ncols);

struct Library *IFFParseBase = NULL;
struct Library *GfxBase = NULL;

/* ILBM frame */
struct ILBMInfo ilbm = {0};


/* ILBM Property chunks to be grabbed - BMHD and CMAP needed for this app
 */
LONG    ilbmprops[] = {
                ID_ILBM, ID_BMHD,
                ID_ILBM, ID_CMAP,
                TAG_DONE
                };

/* ILBM Collection chunks (more than one in file) to be gathered */
LONG    *ilbmcollects = NULL;   /* none needed for this app */

/* ILBM Chunk to stop on */
LONG    ilbmstops[] = {
                ID_ILBM, ID_BODY,
                TAG_DONE
                };


/** main() ******************************************************************/

void main(int argc, char **argv)
    {
    LONG error=NULL;
    UBYTE *ilbmname, fname[80], buf[24];

    if ((argc < 2)||(argv[argc-1][0]=='?'))
        bye("Usage from CLI: 'ILBMtoRaw filename'\n",RETURN_OK);
    if(!(GfxBase = OpenLibrary("graphics.library",0)))
        bye("Can't open graphics.library",RETURN_FAIL);

    if(!(IFFParseBase = OpenLibrary("iffparse.library",0)))
        bye("Can't open iffparse.library",RETURN_FAIL);

/*
 * Here we set up default ILBMInfo fields for our
 * application's frames.
 * Above we have defined the propery and collection chunks
 * we are interested in (some required like BMHD)
 */
    ilbm.ParseInfo.propchks       = ilbmprops;
    ilbm.ParseInfo.collectchks    = ilbmcollects;
    ilbm.ParseInfo.stopchks       = ilbmstops;
    if(!(ilbm.ParseInfo.iff = AllocIFF()))
        bye(IFFerr(IFFERR_NOMEM),RETURN_FAIL);  /* Alloc an IFFHandle */

    ilbmname = argv[1];

    /* Load as a brush since we don't need to display it */
    if (error = loadbrush(&ilbm,ilbmname))
        {
        printf("Can't load ilbm \"%s\", ifferr=%s\n",ilbmname,IFFerr(error));
        bye("",RETURN_WARN);
        }
    else /* Successfully loaded ILBM */
        {
        strcpy(fname,argv[1]);

        if(ilbm.camg & HAM)      strcat(fname, ".ham");
        if(ilbm.camg & EXTRA_HALFBRITE) strcat(fname, ".ehb");

        if(ilbm.camg & HIRES)    strcat(fname, ".hi");
        else strcat(fname, ".lo");

        if(ilbm.camg & LACE)     strcat(fname, ".lace");

        strcat(fname,".");
        sprintf(buf,"%d",ilbm.Bmhd.w);
        strcat(fname,buf);
        strcat(fname,"x");
        sprintf(buf,"%d",ilbm.Bmhd.h);
        strcat(fname,buf);
        strcat(fname,"x");
        sprintf(buf,"%d",ilbm.brbitmap->Depth);
        strcat(fname, buf);
        printf(" Creating file %s \n", fname);
        error=SaveBitMap(fname, ilbm.brbitmap, ilbm.colortable, ilbm.ncolors);

        unloadbrush(&ilbm);
        }

    if(error)   bye(IFFerr(error),RETURN_WARN);
    else        bye("",RETURN_OK);
    }


/* SaveBitMap (as raw planes and colortable)
 *
 * Given filename, bitmap structure, and colortable pointer,
 * writes out raw bitplanes and colortable (not an ILBM)
 * Returns 0 for success
 */
```

```
LONG SaveBitMap(UBYTE *name, struct BitMap *bm, SHORT *cols, int ncols)
    {
    SHORT i;
    LONG nb,plsize;

    LONG file = Open( name, MODE_NEWFILE);
    if( file == 0 )
        {
        printf(" couldn't open %s \n",name);
        return(CLIENT_ERROR);   /* couldnt open a load-file */
        }
    plsize = bm->BytesPerRow*bm->Rows;
    for (i=0; i<bm->Depth; i++)
        {
        nb =  Write(file, bm->Planes[i], plsize);
        if (nb<plsize) break;
        }
    if(nb>0)    nb=Write(file, cols, (1<<bm->Depth)*2); /* save color map */
    Close(file);
    return(nb >= 0 ? 0L : IFFERR_WRITE);
    }

void bye(UBYTE *s, int e)
    {
    if(s&&(*s)) printf("%s\n",s);
    cleanup();
    exit(e);
    }

void cleanup()
    {
    if(ilbm.ParseInfo.iff)              FreeIFF(ilbm.ParseInfo.iff);

    if(IFFParseBase)    CloseLibrary(IFFParseBase);
    if(GfxBase)         CloseLibrary(GfxBase);
    }
```

```
/** Play8SVX.c ***************************************************************
 *
 * Read and play sound sample from an IFF file.  21Jan85
 *
 * By Steve Hayes, Electronic Arts.
 * This software is in the public domain.
 *
 * Modified 05/91 for use with iffparse & to play notes - CAS_CBM
 * requires linkage with several IFF modules - see Makefile
 ****************************************************************************/

#include "iffp/8svxapp.h"

#include <exec/execbase.h>
#include <graphics/gfxbase.h>
#include <clib/alib_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

/* prototypes for our functions */
void cleanup(void);
void bye(UBYTE *s,int error);
void DUnpack(BYTE source[], LONG n, BYTE dest[]);
BYTE D1Unpack(BYTE source[], LONG n, BYTE dest[], BYTE x);
LONG LoadSample(struct EightSVXInfo *esvx, UBYTE *filename);
void UnloadSample(struct EightSVXInfo *esvx);
LONG LoadSBody(struct EightSVXInfo *esvx);
void UnloadSBody(struct EightSVXInfo *esvx);

LONG ShowSample(struct EightSVXInfo *esvx);

LONG OpenAudio(void);
void CloseAudio(void);
LONG PlaySample(struct EightSVXInfo *esvx,
                LONG octave, LONG note, UWORD volume, ULONG delay);

struct IOAudio *playbigsample(struct IOAudio *aio0, struct IOAudio *aio1,
                BYTE *samptr, LONG ssize, ULONG period, UWORD volume);

#define MINARGS 2
char *vers = "\0$VER: Play8SVX 37.5";
char *Copyright = "Play8SVX v37.5 (Freely Redistributable)";
char *usage = "Usage: Play8SVX 8SVXname";


/* globals */
struct Library *IFFParseBase   = NULL;
struct Library *GfxBase = NULL;

BOOL   FromWb;

/* 8SVX Property chunks to be grabbed
 */
LONG    esvxprops[] = {
                ID_8SVX, ID_VHDR,
                ID_8SVX, ID_NAME,
                ID_8SVX, ID_ATAK,
                ID_8SVX, ID_RLSE,
                ID_8SVX, ID_AUTH,
                ID_8SVX, ID_Copyright,
                TAG_DONE
                };
```

```
/* 8SVX Collection chunks (more than one in file) to be gathered */
LONG    esvxcollects[] = {
                ID_8SVX, ID_ANNO,
                TAG_DONE
                };

/* 8SVX Chunk to stop on */
LONG    esvxstops[] = {
                ID_8SVX, ID_BODY,
                TAG_DONE
                };


UBYTE nomem[]  = "Not enough memory\n";
UBYTE noiffh[] = "Can't alloc iff\n";



/* For our allocated EightSVXInfo */
struct EightSVXInfo  *esvx = NULL;


/*
 * MAIN
 */
void main(int argc, char **argv)
   {
   UBYTE *esvxname=NULL;
   ULONG oct;
   LONG error=0L;

   FromWb = argc ? FALSE : TRUE;

   if((argc<MINARGS)||(argv[argc-1][0]=='?'))
        {
        printf("%s\n%s\n",Copyright,usage);
        bye("",RETURN_OK);
        }

   esvxname = argv[1];

/* Open Libraries */
   if(!(IFFParseBase = OpenLibrary("iffparse.library",0)))
      bye("Can't open iffparse library.\n",RETURN_WARN);


/*
 * Alloc one EightSVXInfo struct
 */
    if(!(esvx = (struct EightSVXInfo *)
        AllocMem(sizeof(struct EightSVXInfo),MEMF_PUBLIC|MEMF_CLEAR)))
                bye(nomem,RETURN_FAIL);

/*
 * Here we set up our EightSVXInfo fields for our
 * application.
 * Above we have defined the propery and collection chunks
 * we are interested in (some required like VHDR).
 * We want to stop on BODY.
 */
    esvx->ParseInfo.propchks    = esvxprops;
    esvx->ParseInfo.collectchks = esvxcollects;
    esvx->ParseInfo.stopchks    = esvxstops;
/*
 * Alloc the IFF handle for the frame
 */
    if(!(esvx->ParseInfo.iff = AllocIFF())) bye(noiffh,RETURN_FAIL);


    if(!(error = LoadSample(esvx, esvxname)))
        {
        ShowSample(esvx);

        if(!(error = OpenAudio()))
            {
            /* If we think this is a sound effect, play it as such (note=-1) */
            if((esvx->Vhdr.ctOctave==1)&&(esvx->Vhdr.samplesPerSec)
                &&(esvx->Vhdr.oneShotHiSamples)&&(!esvx->Vhdr.repeatHiSamples))
                {
                PlaySample(esvx,0,-1,64,0);
                }
            /* Else play it like an instrument */
            else
                {
                for(oct=0; oct < esvx->Vhdr.ctOctave; oct++)
                    {
                    PlaySample(esvx,oct,0,64,50);
                    PlaySample(esvx,oct,4,64,50);
                    PlaySample(esvx,oct,7,64,50);
                    }
                }
            CloseAudio();
            }
        else printf("error opening audio device\n");
        }
    else
        printf("%s\n",IFFerr(error));

    cleanup();
    exit(RETURN_OK);
    }


void bye(UBYTE *s,int error)
    {
    if((*s)&&(!FromWb)) printf("%s\n",s);
    cleanup();
    exit(error);
    }


void cleanup()
    {
    if(esvx)
        {
        DD(bug("About to UnloadSample\n"));
        UnloadSample(esvx);

        DD(bug("About to FreeIFF\n"));
        if(esvx->ParseInfo.iff)         FreeIFF(esvx->ParseInfo.iff);

        DD(bug("About to free EightSVXInfo\n"));
        FreeMem(esvx,sizeof(struct EightSVXInfo));
        }

    if(IFFParseBase)     CloseLibrary(IFFParseBase);
    }


/** ShowSample() **********************************************
 *
```

```
 * Show sample information after calling LoadSample()
 *
 *********************************************************************/
LONG ShowSample(struct EightSVXInfo *esvx)
    {
    LONG error = 0L;
    BYTE *buf;
    Voice8Header *vhdr;

    if(!esvx)                    return(CLIENT_ERROR);
    if(!(buf = esvx->sample))    return(CLIENT_ERROR);

    /* LoadSample copied VHDR and NAME (if any) to our esvx frame */
    vhdr = &esvx->Vhdr;
    if(esvx->name[0]) printf("\nNAME: %s",esvx->name);

    printf("\n\nVHDR Info:");
    printf("\noneShotHiSamples=%ld", vhdr->oneShotHiSamples);
    printf("\nrepeatHiSamples=%ld", vhdr->repeatHiSamples);
    printf("\nsamplesPerHiCycle=%ld", vhdr->samplesPerHiCycle);
    printf("\nsamplesPerSec=%ld", vhdr->samplesPerSec);
    printf("\nctOctave=%ld", vhdr->ctOctave);
    printf("\nsCompression=%ld", vhdr->sCompression);
    printf("\nvolume=0x%lx", vhdr->volume);
    printf("\nData = %3ld %3ld %3ld %3ld %3ld %3ld %3ld %3ld",
           buf[0],buf[1],buf[2],buf[3],buf[4],buf[5],buf[6],buf[7]);
    printf("\n       %3ld %3ld %3ld %3ld %3ld %3ld %3ld %3ld ...\n",
           buf[8+0],buf[8+1],buf[8+2],buf[8+3],buf[8+4],buf[8+5],
           buf[8+6],buf[8+ 7]);

    return(error);
    }


/* OpenAudio
 *
 * Opens audio device for one audio channel, 2 IO requests
 * Returns 0 for success
 *
 * Based on code by Dan Baker
 */

UBYTE           whichannel[] = { 1,2,4,8 };

/* periods for scale starting at   65.40Hz (C) with 128 samples per cycle
 *                              or  130.81Hz (C) with  64 samples per cycle
 *                              or  261.63Hz (C) with  32 samples per cycle
 *                              or  523.25Hz (C) with  16 samples per cycle
 *                              or 1046.50Hz (C) with   8 samples per cycle
 *                              or 2093.00Hz (C) with   4 samples per cycle
 */

UWORD   per_ntsc[12]= { 428, 404, 380, 360,
                        340, 320, 302, 286,
                        270, 254, 240, 226 };

/* periods adjusted for system clock frequency */
UWORD   per[12];

/* Note - these values 3579545 NTSC, 3546895 PAL */
#define NTSC_CLOCK 3579545L
#define PAL_CLOCK  3546895L

#define AIOCNT 4
struct  IOAudio *aio[AIOCNT] = {NULL};    /* Ptrs to IO blocks for commands  */
struct  MsgPort *port;          /* Pointer to a port so the device can reply */
BOOL    devopened;
ULONG   clock = NTSC_CLOCK;     /* Will check for PAL and change if necessary */


LONG OpenAudio()
{
extern  struct ExecBase *SysBase;
LONG    error=0L;
ULONG   period;
int     k;

if(devopened)   return(-1);

/*------------------------------------------------------------------------*/
/* Ask the system if we are PAL or NTSC and set clock constant accordingly */
/*------------------------------------------------------------------------*/
if(GfxBase=OpenLibrary("graphics.library",0L))
    {
    if(((struct GfxBase *)GfxBase)->DisplayFlags & PAL)
                clock = PAL_CLOCK;
    else
                clock = NTSC_CLOCK;
    CloseLibrary((struct Library *) GfxBase);
    }

printf("OpenAudio: For period calculations, clock=%ld\n", clock);

/* calculate period values for one octave based on system clock */
for(k=0; k<12; k++)
    {
    period = ((per_ntsc[k] * clock) + (NTSC_CLOCK >> 1)) / NTSC_CLOCK;
    per[k] = period;
    D(bug("per[%ld]=%ld ",k,per[k]));
    }
D(bug("\n"));

/*------------------------------------------------------------------*/
/* Create a reply port so the audio device can reply to our commands */
/*------------------------------------------------------------------*/
if(!(port=CreatePort(0,0)))
        { error = 1; goto bailout; }

/*------------------------------------------------------------------------*/
/*  Create audio I/O blocks so we can send commands to the audio device     */
/*------------------------------------------------------------------------*/
for(k=0; k<AIOCNT; k++)
    {
    if(!(aio[k]=(struct IOAudio *)CreateExtIO(port,sizeof(struct IOAudio))))
        { error = k+2; goto bailout; }
    }

/*--------------------------------------------------------------------*/
/* Set up the audio I/O block for channel allocation:                 */
/* ioa_Request.io_Message.mn_ReplyPort is the address of a reply port.  */
/* ioa_Request.io_Message.mn_Node.ln_Pri sets the precedence (priority) */
/*   of our use of the audio device. Any tasks asking to use the audio  */
/*   device that have a higher precedence will steal the channel from us.*/
/* ioa_Request.io_Command is the command field for IO.                */
/* ioa_Request.io_Flags is used for the IO flags.                     */
/* ioa_AllocKey will be filled in by the audio device if the allocation */
/*   succeeds. We must use the key it gives for all other commands sent.*/
/* ioa_Data is a pointer to the array listing the channels we want.     */
/* ioa_Length tells how long our list of channels is.                 */
/*--------------------------------------------------------------------*/
aio[0]->ioa_Request.io_Command              = ADCMD_ALLOCATE;
```

```
aio[0]->ioa_Request.io_Flags                = ADIOF_NOWAIT;
aio[0]->ioa_AllocKey                        = 0;
aio[0]->ioa_Data                            = whichannel;
aio[0]->ioa_Length                          = sizeof(whichannel);

/*-----------------------------------------------*/
/* Open the audio device and allocate a channel  */
/*-----------------------------------------------*/
if(!(OpenDevice("audio.device",0L, (struct IORequest *) aio[0] ,0L)))
        devopened = TRUE;
else { error = 5; goto bailout; }

/* Clone the flags, channel allocation, etc. into other IOAudio requests */
for(k=1; k<AIOCNT; k++) *aio[k] = *aio[0];

bailout:
if(error)
    {
    printf("OpenAudio errored out at step %ld\n",error);
    CloseAudio();
    }
return(error);
}


/* CloseAudio
 *
 * Close audio device as opened by OpenAudio, null out pointers
 */
void CloseAudio()
{
int k;

D(bug("Closing audio device...\n"));

/* Note - we know we have no outstanding audio requests */
if(devopened)
    {
    CloseDevice((struct IORequest *) aio[0]);
    devopened = FALSE;
    }

for(k=0; k<AIOCNT; k++)
    {
    if(aio[k])  DeleteExtIO(aio[k]), aio[k] = NULL;
    }

if(port)        DeletePort(port),  port = NULL;
}


/** PlaySample() ***********************************************
 *
 * Play a note in octave for delay/50ths of a second
 * OR Play a sound effect (set octave and note to 0, -1)
 *
 * Requires successful OpenAudio() called previously
 *
 * When playing notes:
 * Expects note values between 0 (C) and 11 (B#)
 * Uses largest octave sample in 8SVX as octave 0, next smallest
 *   as octave 1, etc.
 *
 * Notes - this simple example routine does not do ATAK and RLSE)
 *       - use of Delay for timing is simplistic, synchronous, and does
 *             not take into account that the oneshot itself may be
 *              longer than the delay.
 *         Use timer.device for more accurate asynchronous delays
 *
 ***********************************************************************/
/* Max playable sample in one IO request is 128K */
#define MAXSAMPLE 131072

LONG    PlaySample(struct EightSVXInfo *esvx,
                        LONG octave, LONG note, UWORD volume, ULONG delay)
{
/* pointers to outstanding requests */
struct          IOAudio *aout0=NULL, *aout1=NULL;
ULONG           period;
LONG            osize, rsize;
BYTE            *oneshot, *repeat;

if(!devopened)  return(-1);

if(note > 11) note=0;

if( note == -1 ) period = clock / esvx->Vhdr.samplesPerSec;
else             period = per[note]; /* table set up by OpenAudio */

if(octave > esvx->Vhdr.ctOctave) octave = 0;
if(volume > 64) volume = 64;

oneshot  = esvx->osamps[octave];
osize    = esvx->osizes[octave];
repeat   = esvx->rsamps[octave];
rsize    = esvx->rsizes[octave];

D(bug("oneshot $%lx size %ld, repeat $%lx size %ld\n",
        oneshot, osize, repeat, rsize));

/*-------------------------------------------------------------*/
/* Set up audio I/O blocks to play a sample using CMD_WRITE.  */
/* Set up one request for the oneshot and one for repeat      */
/* (all ready for simple case, but we may not need both)      */
/* The io_Flags are set to ADIOF_PERVOL so we can set the     */
/*    period (speed) and volume with the our sample;          */
/* ioa_Data points to the sample; ioa_Length gives the length */
/* ioa_Cycles tells how many times to repeat the sample       */
/* If you want to play the sample at a given sampling rate,   */
/* set ioa_Period = clock/(given sampling rate)               */
/*-------------------------------------------------------------*/
aio[0]->ioa_Request.io_Command                =CMD_WRITE;
aio[0]->ioa_Request.io_Flags                  =ADIOF_PERVOL;
aio[0]->ioa_Data                              =oneshot;
aio[0]->ioa_Length                            =osize;
aio[0]->ioa_Period                            =period;
aio[0]->ioa_Volume                            =volume;
aio[0]->ioa_Cycles                            =1;

aio[2]->ioa_Request.io_Command                =CMD_WRITE;
aio[2]->ioa_Request.io_Flags                  =ADIOF_PERVOL;
aio[2]->ioa_Data                              =repeat;
aio[2]->ioa_Length                            =rsize;
aio[2]->ioa_Period                            =period;
aio[2]->ioa_Volume                            =volume;
aio[2]->ioa_Cycles                            =0;  /* repeat until stopped */

/*-------------------------------------------------*/
/* Send the command to start a sound using BeginIO() */
/* Go to sleep and wait for the sound to finish with */
/* WaitIO() to wait and get the get the ReplyMsg     */
/*-------------------------------------------------*/
```

```
printf("Starting tone O len %ld for %01d cyc, R len %ld for %01d cyc, per=%ld...",
                osize, aio[0]->ioa_Cycles, rsize, aio[1]->ioa_Cycles, period);

if(osize)
    {
    /* Simple case for oneshot sample <= 128K (ie. most samples) */
    if(osize <= MAXSAMPLE)      BeginIO((struct IORequest *)(aout0=aio[0]));

    /* Note - this else case code is for samples >128K */
    else
        {
        *aio[1] = *aio[0];
        aout0 = playbigsample(aio[0],aio[1],oneshot,osize,period,volume);
        }
     }

if(rsize)
    {
    /* Simple case for oneshot sample <= 128K (ie. most samples) */
    if(rsize <= MAXSAMPLE)      BeginIO((struct IORequest *)(aout1=aio[2]));

    /* Note - this else case code is for samples >128K */
    else
        {
        *aio[3] = *aio[2];
        aout1 = playbigsample(aio[2],aio[3],repeat,rsize,period,volume);
        }
     }

if(delay)       Delay(delay);   /* crude timing for notes */

/* Wait for any requests we still have out */
if(aout0) WaitIO(aout0);

if(aout1)
   {
   if(note >= 0) AbortIO(aout1);        /* if a note, stop it now */
   WaitIO(aout1);
   }

printf("Done\n");
}


/** playbigsample() ***********************************************************
 *
 *  called by playsample to deal with samples > 128K
 *
 *  wants pointers to two ready-to-use IOAudio iorequest blocks
 *
 *  returns pointer to the IOAudio request that is still out
 *   or NULL if none (error)
 ***************************************************************************/

struct IOAudio *playbigsample(struct IOAudio *aio0, struct IOAudio* aio1,
                        BYTE *samptr, LONG ssize, ULONG period, UWORD volume)
{
struct IOAudio *aio[2];
LONG   size;
int    req=0, reqn=1;   /* current and next IOAudio request indexes */

if((!aio0)||(!aio1)||(ssize < MAXSAMPLE))       return(NULL);

aio[req]  = aio0;
aio[reqn] = aio1;
```

```
/* start the first 128 K playing */
aio[req]->ioa_Request.io_Command           =CMD_WRITE;
aio[req]->ioa_Request.io_Flags             =ADIOF_PERVOL;
aio[req]->ioa_Data                         =samptr;
aio[req]->ioa_Length                       =MAXSAMPLE;
aio[req]->ioa_Period                       =period;
aio[req]->ioa_Volume                       =volume;
aio[req]->ioa_Cycles                       =1;
BeginIO((struct IORequest*)aio[req]);

for(samptr=samptr + MAXSAMPLE, size = ssize - MAXSAMPLE;
        size > 0;
                samptr += MAXSAMPLE)
    {
    /* queue the next piece of sample */
    reqn = req ^ 1;     /* alternate IO blocks 0 and 1 */
    aio[reqn]->ioa_Request.io_Command            =CMD_WRITE;
    aio[reqn]->ioa_Request.io_Flags              =ADIOF_PERVOL;
    aio[reqn]->ioa_Data                          =samptr;
    aio[reqn]->ioa_Length = (size > MAXSAMPLE) ? MAXSAMPLE : size;
    aio[reqn]->ioa_Period                        =period;
    aio[reqn]->ioa_Volume                        =volume;
    aio[reqn]->ioa_Cycles                        =1;
    BeginIO((struct IORequest*)aio[reqn]);

    /* Wait for previous request to finish */
    WaitIO(aio[req]);
    /* decrement size */
    size = (size > MAXSAMPLE) ? size-MAXSAMPLE : 0;
    req = reqn;          /* switch between aio[0] and aio[1] */
    }
return(aio[reqn]);
}

/** LoadSample() *********************************************************
 *
 * Read 8SVX, given an initialized EightSVXInfo with not-in-use IFFHandle,
 *   and filename.  Leaves the IFFHandle open so you can FindProp()
 *   additional chunks or copychunks().  You must UnloadSample()
 *   when done.  UnloadSample will closeifile if the file is still
 *   open.
 *
 * Fills in esvx->Vhdr and Name, and allocates/loads esvx->sample,
 *   setting esvx->samplebytes to size for deallocation.
 *
 * Returns 0 for success of an IFFERR (libraries/iffparse.h)
 *************************************************************************/
LONG LoadSample(struct EightSVXInfo *esvx, UBYTE *filename)
    {
    struct IFFHandle *iff;
    struct StoredProperty *sp;
    Voice8Header *vhdr;
    BYTE *oneshot, *repeat;
    ULONG osize, rsize, spcyc;
    int oct;
    LONG error = 0L;

    D(bug("LoadSample:\n"));

    if(!esvx)                           return(CLIENT_ERROR);
    if(!(iff=esvx->ParseInfo.iff))      return(CLIENT_ERROR);

    if(!(error = openifile((struct ParseInfo *)esvx, filename, IFFF_READ)))
        {
        printf("Reading '%s'...\n",filename);
        error = parseifile((struct ParseInfo *)esvx,
```

```
                ID_FORM, ID_8SVX,
                esvx->ParseInfo.propchks,
                esvx->ParseInfo.collectchks,
                esvx->ParseInfo.stopchks);

        D(bug("LoadSample: after parseifile - error = %ld\n",error));

        if((!error)||(error == IFFERR_EOC)||(error == IFFERR_EOF))
            {
            if(contextis(iff,ID_8SVX,ID_FORM))
                {
                D(bug("LoadSample: context is 8SVX\n"));
                if(!(sp = FindProp(iff,ID_8SVX,ID_VHDR)))
                    {
                    message("No 8SVX.VHDR!");
                    error = IFFERR_SYNTAX;
                    }
                else
                    {
                    D(bug("LoadSample: Have VHDR\n"));
                    /* copy Voice8Header into frame */
                    vhdr = (Voice8Header *)(sp->sp_Data);
                    *(&esvx->Vhdr) = *vhdr;
                    /* copy name if any */
                    esvx->name[0]='\0';
                    if(sp = FindProp(iff,ID_8SVX,ID_NAME))
                        {
                        strncpy(esvx->name,sp->sp_Data,sp->sp_Size);
                        esvx->name[MIN(sp->sp_Size,79)] = '\0';
                        }
                    error = LoadSBody(esvx);
                    D(bug("LoadSample: After LoadSBody - error = %ld\n",error));
                    if(!error)
                        {
                        osize   = esvx->Vhdr.oneShotHiSamples;
                        rsize   = esvx->Vhdr.repeatHiSamples;
                        spcyc   = esvx->Vhdr.samplesPerHiCycle;
                        if(!spcyc)  spcyc = esvx->Vhdr.repeatHiSamples;
                        if(!spcyc)  spcyc = 8;

                        oneshot = esvx->sample;

                        for(oct = esvx->Vhdr.ctOctave-1; oct >= 0;
                                oct--, oneshot+=(osize+rsize),
                                       osize <<= 1, rsize <<=1, spcyc <<=1)
                            {
                            repeat  = oneshot + osize;
                            esvx->osizes[oct] = osize;
                            if(osize)  esvx->osamps[oct] = oneshot;
                            else       esvx->osamps[oct] = 0;
                            esvx->rsizes[oct] = rsize;
                            if(rsize)  esvx->rsamps[oct] = repeat;
                            else       esvx->rsamps[oct] = 0;
                            esvx->spcycs[oct] = spcyc;

                        D(bug("oneshot $%lx size %ld, repeat $%lx size %ld\n",
                                oneshot, osize, repeat, rsize));

                            }
                        }
                    }
                }
            else
                {
                message("Not an 8SVX\n");
                error = NOFILE;
```

```
                    }
                }
            }

        if(error)
            {
            closeifile((struct ParseInfo *)esvx);
            UnloadSample(esvx);
            }
        return(error);
        }


/** UnloadSample() *******************************************************
 *
 * Frees and closes everything opened/alloc'd by LoadSample()
 *
 *************************************************************************/
void UnloadSample(struct EightSVXInfo *esvx)
    {
    if(esvx)
        {
        UnloadSBody(esvx);
        closeifile((struct ParseInfo *)esvx);
        }
    }


/** LoadSBody() **********************************************************
 *
 * Read a 8SVX Sample BODY into RAM.
 *
 *************************************************************************/
LONG LoadSBody(struct EightSVXInfo *esvx)
    {
    struct IFFHandle *iff;
    LONG sbytes, rlen, error = 0L;
    ULONG memtype;
    Voice8Header *vhdr = &esvx->Vhdr;
    BYTE *t;

    D(bug("LoadSBody:\n"));

    if(!(iff=esvx->ParseInfo.iff))      return(CLIENT_ERROR);
    if(!esvx)                           return(CLIENT_ERROR);

    if(!(currentchunkis(iff,ID_8SVX,ID_BODY)))
        {
        message("LoadSBody: not at BODY!");
        return(IFFERR_READ);
        }

    sbytes  = ChunkMoreBytes(CurrentChunk(iff));

    /* if we have to decompress, let's just load it into public mem */
    memtype = vhdr->sCompression ? MEMF_PUBLIC : MEMF_CHIP;

    D(bug("LoadSBody: samplebytes=%ld, compression=%ld\n",
                    sbytes,vhdr->sCompression));

    if(!(esvx->sample = (BYTE *)AllocMem(sbytes, memtype)))
        {
        error = CLIENT_ERROR;
        }
    else
        {
```

```
        D(bug("LoadSBody: have load buffer\n"));
        esvx->samplebytes = sbytes;
        if(rlen=ReadChunkBytes(iff,esvx->sample,sbytes) != sbytes)
            error = IFFERR_READ;

        if(error)
            {
            D(bug("LoadSBody: ReadChunkBytes error = %ld, read %ld bytes\n",
                        error));
            UnloadSample(esvx);
            }
        else if (vhdr->sCompression) /* Decompress, if needed. */
            {
            if(t = (BYTE *)AllocMem(sbytes<<1, MEMF_CHIP))
                {
                D(bug("LoadSBody: have decompression buffer\n"));
                DUnpack(esvx->sample, sbytes, t);
                FreeMem(esvx->sample, sbytes);
                esvx->sample = t;
                esvx->samplebytes = sbytes << 1;
                }
            else
                {
                UnloadSample(esvx);
                error = IFFERR_NOMEM;
                }
            }
        }
    return(error);
    }


/** UnloadSBody() ********************************************************
 *
 * Deallocates esvx->smaple
 *
 *************************************************************************/
void UnloadSBody(struct EightSVXInfo *esvx)
    {
    if(esvx)
        {
        if(esvx->sample)
            {
            DD(bug("About to free SBody\n"));
            FreeMem(esvx->sample,esvx->samplebytes);
            esvx->sample = NULL;
            }
        esvx->samplebytes = NULL;
        }
    }


/* DUnpack.c --- Fibonacci Delta decompression by Steve Hayes */

/* Fibonacci delta encoding for sound data */
BYTE codeToDelta[16] = {-34,-21,-13,-8,-5,-3,-2,-1,0,1,2,3,5,8,13,21};

/* Unpack Fibonacci-delta encoded data from n byte source
 * buffer into 2*n byte dest buffer, given initial data
 * value x.  It returns the lats data value x so you can
 * call it several times to incrementally decompress the data.
 */

BYTE D1Unpack(BYTE source[], LONG n, BYTE dest[], BYTE x)
   {
   BYTE d;
   LONG i, lim;

   lim = n << 1;
   for (i=0; i < lim; ++i)
      {
      /* Decode a data nibble, high nibble then low nibble */
      d = source[i >> 1];     /* get a pair of nibbles */
      if (i & 1)              /* select low or high nibble */
         d &= 0xf;            /* mask to get the low nibble */
      else
         d >>= 4;             /* shift to get the high nibble */
      x += codeToDelta[d];    /* add in the decoded delta */
      dest[i] = x;            /* store a 1 byte sample */
      }
   return(x);
   }

/* Unpack Fibonacci-delta encoded data from n byte
 * source buffer into 2*(n-2) byte dest buffer.
 * Source buffer has a pad byte, an 8-bit initial
 * value, followed by n-2 bytes comprising 2*(n-2)
 * 4-bit encoded samples.
 */

void DUnpack(source, n, dest)
BYTE source[], dest[];
LONG n;
   {
   D1Unpack(source+2, n-2, dest, source[1]);
   }
```

```
/* RawtoILBM
 * Converts raw file (from ILBMtoRaw) into an ILBM
 * Requires linkage with several iffparse modiules - See Makefile
 */

#include "iffp/ilbmapp.h"

#include <intuition/intuitionbase.h>
#include <workbench/workbench.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

char *vers = "\0$VER: RawtoILBM 37.5";
char *Copyright =
  "RawtoILBM v37.5 - converts raw file to ILBM - Freely Redistributable";
#define MINARGS 6
char *usage = "Usage: RawtoILBM rawname ilbmname width height depth\n";

void bye(UBYTE *s,int e);
void cleanup(void);

struct Library  *IntuitionBase = NULL;
struct Library  *GfxBase = NULL;
struct Library  *IFFParseBase = NULL;

struct ILBMInfo ilbm = {0};

USHORT  colortable[MAXAMCOLORREG];

BOOL fromWB;


void main(int argc, char **argv)
    {
    LONG        error = 0L, rawfile, rlen;
    USHORT      width, height, depth, pwidth, pheight, pmode, extra;
    ULONG       plsize;
    char        *rawname,*ilbmname;
    int         k;

    fromWB = (argc==0) ? TRUE : FALSE;

    if(!(IntuitionBase = OpenLibrary("intuition.library", 0)))
      bye("Can't open intuition library.\n",RETURN_WARN);

    if(!(GfxBase = OpenLibrary("graphics.library",0)))
      bye("Can't open graphics library.\n",RETURN_WARN);

    if(!(IFFParseBase = OpenLibrary("iffparse.library",0)))
      bye("Can't open iffparse library.\n",RETURN_WARN);

    if(!(ilbm.ParseInfo.iff = AllocIFF()))
      bye(IFFerr(IFFERR_NOMEM),RETURN_WARN);

    if(argc==MINARGS)                 /* Passed filenames via command line  */
        {
        rawname  = argv[1];
        ilbmname = argv[2];
        width  = atoi(argv[3]);
        height  = atoi(argv[4]);
        depth  = atoi(argv[5]);

        /* Page width, height, and mode for saved ILBM */
        pwidth  = width  < 320 ? 320 : width;
        pheight = height < 200 ? 200 : height;
        pmode   = pwidth >= 640  ? HIRES : 0L;
        pmode  |= pheight >= 400 ? LACE  : 0L;

        plsize = RASSIZE(width,height);
        }
    else
        {
        printf("%s\n%s\n",Copyright,usage);
        bye("\n",RETURN_OK);
        }


    if(!(rawfile = Open(rawname,MODE_OLDFILE)))
        {
        printf("Can't open raw file '%s'\n",rawname);
        bye(" ",RETURN_WARN);
        }

    /*
     * Allocate Bitmap and planes
     */
     extra = depth > 8 ? depth - 8 : 0;
     if(ilbm.brbitmap = AllocMem(sizeof(struct BitMap) + (extra<<2),
                                MEMF_CLEAR))
        {
        InitBitMap(ilbm.brbitmap,depth,width,height);
        for(k=0, error=0, rlen=1; k<depth && (!error) && (rlen >0); k++)
            {
            if(!(ilbm.brbitmap->Planes[k] = AllocRaster(width,height)))
                        error = IFFERR_NOMEM;
            if(! error)
                {
                BltClear(ilbm.brbitmap->Planes[k], RASSIZE(width,height),0);
                /* Read a plane */
                rlen = Read(rawfile,ilbm.brbitmap->Planes[k],plsize);
                }
            }

        /* get colortable */
        if((!error)&&(rlen > 0))
                rlen=Read(rawfile,colortable,(MIN(1<<depth,MAXAMCOLORREG)<<1));

        if((error)||(rlen<=0))
            {
            if(rlen <= 0)       printf("Error loading raw file - check dimensions\n
");
            else                printf("Error allocating planes\n");
            }
        else
            {
            error = saveilbm(&ilbm, ilbm.brbitmap, pmode,
                width,  height, pwidth, pheight,
                colortable, MIN(1<<depth,MAXAMCOLORREG), 4,    /* colors */
                mskNone, 0,             /* masking. transColor */
                NULL, NULL,             /* additional chunk lists */
                ilbmname);
            }

        for(k=0; k<depth; k++)
            {
            if(ilbm.brbitmap->Planes[k])
                        FreeRaster(ilbm.brbitmap->Planes[k],width,height);
            }
        FreeMem(ilbm.brbitmap, sizeof(struct BitMap) + (extra << 2));
```

```
        }

    Close(rawfile);

    if(error)
        {
        printf("%s\n",IFFerr(error));
        bye(" ", RETURN_FAIL);
        }
    else bye("",RETURN_OK);
    }


void bye(UBYTE *s,int e)
    {
    if(s&&(*s)) printf("%s\n",s);
    if ((fromWB)&&(*s))    /* Wait so user can read messages */
        {
        printf("\nPRESS RETURN TO EXIT\n");
        while(getchar() != '\n');
        }
    cleanup();
    exit(e);
    }

void cleanup()
    {
    if(ilbm.ParseInfo.iff)      FreeIFF(ilbm.ParseInfo.iff);

    if(GfxBase)         CloseLibrary(GfxBase);
    if(IntuitionBase)   CloseLibrary(IntuitionBase);
    if(IFFParseBase)    CloseLibrary(IFFParseBase);
    }
```

```
/* ScreenSave
 * Saves front screen as an ILBM
 * Requires linkage with several iffparse modiules - See Makefile
 */

#include "iffp/ilbmapp.h"

#include <intuition/intuitionbase.h>
#include <workbench/workbench.h>

#include <clib/icon_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

char *vers = "\0$VER: screensave 37.5";
char *Copyright =
  "screensave v37.5 - supports new modes - Freely Redistributable";
char *usage =
 "Usage: screensave filename (filename -c[unit] for clipboard)\n"
 "Options: QUIET, NODELAY, NOICON, SEQUENCE (sequence adds a number to name)\n"
 "Saves front screen after 10-sec delay (unless NODELAY).\n";

int mygets(char *s);
void bye(UBYTE *s,int e);
void cleanup(void);

struct Library  *IntuitionBase = NULL;
struct Library  *GfxBase = NULL;
struct Library  *IconBase = NULL;
struct Library  *IFFParseBase = NULL;

struct ILBMInfo ilbm = {0};

BOOL fromWB, Quiet, NoDelay, NoIcon, Sequence;



#define INBUFSZ 128
char nbuf[INBUFSZ];

/* Data for project icon for saved ILBM
 *
 */
UWORD ILBMI1Data[] =
{
/* Plane 0 */
    0x0000,0x0000,0x0000,0x0001,0x0000,0x0000,0x0000,0x0003,
    0x0FFF,0xFFFF,0xFFFF,0xFFF3,0x0FFF,0x0000,0x0000,0xFFF3,
    0x0FFC,0x0000,0x0000,0x3FF3,0x0FE0,0x0E80,0xF800,0x07F3,
    0x0F80,0x1C01,0x8C00,0x01F3,0x0F00,0x0001,0x8C00,0x00F3,
    0x0600,0x0000,0x0600,0x0063,0x0600,0x0003,0xBC00,0x0063,
    0x0600,0x0001,0xFC00,0x0063,0x0600,0x0000,0xFC00,0x0063,
    0x0600,0x1FC1,0xFE40,0x0063,0x0600,0x1DC1,0xFE20,0x0063,
    0x0600,0x1CE3,0xFF12,0x0063,0x0F00,0x1CE0,0x004F,0xC0F3,
    0x0F80,0x1CE0,0x002F,0x01F3,0x0FE0,0x0E78,0x423D,0x07F3,
    0x0FFC,0x0000,0x0000,0x3FF3,0x0FFF,0x0000,0x0000,0xFFF3,
    0x0FFF,0xFFFF,0xFFFF,0xFFF3,0x0000,0x0000,0x0000,0x0003,
    0x7FFF,0xFFFF,0xFFFF,0xFFFF,
/* Plane 1 */
    0xFFFF,0xFFFF,0xFFFF,0xFFFE,0xD555,0x5555,0x5555,0x5554,
    0xD000,0x0000,0x0000,0x0004,0xD3FC,0xFFFF,0xFFFF,0x3FC4,
    0xD3C0,0x0000,0x0000,0x03C4,0xD300,0x0100,0xF800,0x00C4,
    0xD300,0x0381,0xFC00,0x00C4,0xD080,0x0701,0xFC00,0x0104,
```

```
    0xD180,0xF883,0xFE00,0x0194,0xD181,0xDF80,0x4700,0x0194,
    0xD181,0xDF82,0x0180,0x0194,0xD180,0x6F82,0x00C0,0x0194,
    0xD180,0x0002,0x0020,0x0194,0xD180,0x0000,0x0000,0x0194,
    0xD180,0x0000,0x0002,0x0194,0xD080,0x0000,0xCC46,0xC104,
    0xD300,0x0000,0xCC2F,0x00C4,0xD300,0x0E78,0x883D,0x00C4,
    0xD3C0,0x0000,0x0000,0x03C4,0xD3FC,0xFFFF,0xFFFF,0x3FC4,
    0xD000,0x0000,0x0000,0x0004,0xD555,0x5555,0x5555,0x5554,
    0x8000,0x0000,0x0000,0x0000,
};

struct Image ILBMI1 =
{
    0, 0,                           /* Upper left corner */
    64, 23, 2,                      /* Width, Height, Depth */
    ILBMI1Data,                     /* Image data */
    0x0003, 0x0000,                 /* PlanePick, PlaneOnOff */
    NULL                            /* Next image */
};

UBYTE *ILBMTools[] =
{
    "FILETYPE=ILBM",
    NULL
};

struct DiskObject ILBMobject =
{
    WB_DISKMAGIC,                   /* Magic Number */
    WB_DISKVERSION,                 /* Version */
    {                               /* Embedded Gadget Structure */
        NULL,                       /* Next Gadget Pointer */
        0, 0, 64, 24,               /* Left,Top,Width,Height */
        GADGIMAGE | GADGBACKFILL,           /* Flags */
        RELVERIFY | GADGIMMEDIATE,                  /* Activation Flags */
        BOOLGADGET,                 /* Gadget Type */
        (APTR)&ILBMI1,      /* Render Image */
        NULL,                       /* Select Image */
        NULL,                       /* Gadget Text */
        NULL,                       /* Mutual Exclude */
        NULL,                       /* Special Info */
        0,                          /* Gadget ID */
        NULL,                       /* User Data */
    },
    WBPROJECT,                      /* Icon Type */
    "Display",                      /* Default Tool */
    ILBMTools,                      /* Tool Type Array */
    NO_ICON_POSITION,               /* Current X */
    NO_ICON_POSITION,               /* Current Y */
    NULL,                           /* Drawer Structure */
    NULL,                           /* Tool Window */
    0                               /* Stack Size */
};



void main(int argc, char **argv)
   {
   struct Screen   *frontScreen;
   LONG            error = 0L, seqlock;
   char            *filename;
   int l, k;

   fromWB = (argc==0) ? TRUE : FALSE;


   if(!(IntuitionBase = OpenLibrary("intuition.library", 0)))
   bye("Can't open intuition library.\n",RETURN_WARN);

if(!(GfxBase = OpenLibrary("graphics.library",0)))
   bye("Can't open graphics library.\n",RETURN_WARN);

if(!(IFFParseBase = OpenLibrary("iffparse.library",0)))
   bye("Can't open iffparse library.\n",RETURN_WARN);

if(!(IconBase = OpenLibrary("icon.library",0)))
   bye("Can't open icon library.\n",RETURN_WARN);

if(!(ilbm.ParseInfo.iff = AllocIFF()))
   bye(IFFerr(IFFERR_NOMEM),RETURN_WARN);

if(argc>1)                  /* Passed filename via command line  */
   {
   if(argv[1][0]=='?')
         {
         printf("%s\n%s\n",Copyright,usage);
         bye("\n",RETURN_OK);
         }
   else filename = argv[1];

   NoDelay = NoIcon = Quiet = Sequence = FALSE;
   for(k=2; k < (argc); k++)
     {
     if(!(stricmp(argv[k],"NODELAY")))       NoDelay  = TRUE;
     else if(!(stricmp(argv[k],"NOICON")))   NoIcon   = TRUE;
     else if(!(stricmp(argv[k],"QUIET")))    Quiet    = TRUE;
     else if(!(stricmp(argv[k],"SEQUENCE"))) Sequence = TRUE;
     }
   if(Sequence)
     {
     for(k=1; k<9999; k++)
         {
         sprintf(nbuf,"%s%04ld",filename,k);
         if(seqlock = Lock(nbuf,ACCESS_READ))        UnLock(seqlock);
         else break;
         }
     filename = nbuf;
     }
   }
else
   {
   printf("%s\n%s\n",Copyright,usage);
   printf("Enter filename for save: ");
   l = mygets(&nbuf[0]);

   if(l==0)                 /* No filename - Exit */
      {
      bye("\nScreen not saved, filename required\n",RETURN_FAIL);
      }
   else
      {
      filename = &nbuf[0];
      }
   }

if(!NoDelay) Delay(500);

Forbid();
frontScreen  = ((struct IntuitionBase *)IntuitionBase)->FirstScreen;
Permit();

if(error = screensave(&ilbm, frontScreen,
                        NULL, NULL,
```

```
                        filename))
      {
      printf("%s\n",IFFerr(error));
      }
   else
      {
      if(!Quiet) printf("Screen saved as %s... ",filename);
      if((!NoIcon)&&(filename[0]!='-')&&(filename[1]!='c')) /* not clipboard */
         {
         if(!(PutDiskObject(filename,&ILBMobject)))
            {
            bye("Error saving icon\n",RETURN_WARN);
            }
         if(!Quiet) printf("Icon saved\n");
         }
      else if(!Quiet) printf("\n");
      bye("",RETURN_OK);
      }
   }


void bye(UBYTE *s,int e)
   {
   if(s&&(*s)) printf("%s\n",s);
   if ((fromWB)&&(*s))    /* Wait so user can read messages */
      {
      printf("\nPRESS RETURN TO EXIT\n");
      mygets(&nbuf[0]);
      }
   cleanup();
   exit(e);
   }

void cleanup()
   {
   if(ilbm.ParseInfo.iff)       FreeIFF(ilbm.ParseInfo.iff);

   if(GfxBase)          CloseLibrary(GfxBase);
   if(IntuitionBase)    CloseLibrary(IntuitionBase);
   if(IconBase)         CloseLibrary(IconBase);
   if(IFFParseBase)     CloseLibrary(IFFParseBase);
   }


int mygets(char *s)
   {
   int l = 0, max = INBUFSZ - 1;

   while (((*s = getchar()) !='\n' )&&(l < max)) s++, l++;
   *s = NULL;
   return(l);
   }
```

modules/bmprintc.c Page 1

```c
/*----------------------------------------------------------------*/
/*                                                                */
/*                        bmprintc.c                              */
/*                                                                */
/* print out a C-language representation of data for bitmap      */
/*                                                                */
/* By Jerry Morrison and Steve Shaw, Electronic Arts.             */
/* This software is in the public domain.                         */
/*                                                                */
/* This version for the Commodore-Amiga computer.                 */
/* Cleaned up and modified a bit by Chuck McManis, Aug 1988       */
/* Modified 05/91 by CBM for use with iffparse modules            */
/*                                                                */
/*----------------------------------------------------------------*/

#include "iffp/ilbmapp.h"
#include <stdio.h>

#define NO  0
#define YES 1

void PSprite(struct BitMap *bm, FILE *fp, UBYTE *name, int p, BOOL dohead);
void PrCRLF(FILE *fp);
void PrintBob(struct BitMap *bm, FILE *fp, UBYTE *name);
void PrintSprite(struct BitMap *bm, FILE *fp, UBYTE *name,
                 BOOL attach, BOOL dohdr);

static BOOL doCRLF;
char sp_colors[] = ".oO@";

void PrCRLF(FILE *fp)
{
        if (doCRLF)
                fprintf(fp, "%c%c", 0xD, 0xA);
        else
                fprintf(fp, "\n");
}

void PrintBob(struct BitMap *bm, FILE *fp, UBYTE *name)
{
        register UWORD  *wp;    /* Pointer to the bitmap data */

        short   p,i,j,nb;       /* temporaries */
        short   nwords = (bm->BytesPerRow/2)*bm->Rows;

        fprintf(fp, "/*----- bitmap : w = %ld, h = %ld ------ */",
                        bm->BytesPerRow*8, bm->Rows);

        PrCRLF(fp);

        for (p = 0; p < bm->Depth; ++p) {       /* For each bit plane */
                wp = (UWORD *)bm->Planes[p];
                fprintf(fp, "/*------ plane # %ld: --------*/", p);
                PrCRLF(fp);
                fprintf(fp, "UWORD %s%c[%ld] = { ", name, (p?('0'+p):' '), nwords);

                PrCRLF(fp);
                for (j = 0; j < bm->Rows; j++, wp += (bm->BytesPerRow >> 1)) {
                        fprintf(fp, "   ");
                        for (nb = 0; nb < (bm->BytesPerRow) >> 1; nb++)
                                fprintf(fp, "0x%04x,", *(wp+nb));
                        if (bm->BytesPerRow <= 6) {
                                fprintf(fp, "\t/* ");
                                for (nb = 0; nb < (bm->BytesPerRow) >> 1; nb++)
                                        for (i=0 ; i<16; i++)
                                                fprintf(fp, "%c",
```

modules/bmprintc.c Page 2

```c
                                                        (((*(wp+nb)>>(15-i))&1) ? '*' : '.'));
                                fprintf(fp, " */");
                        }
                        PrCRLF(fp);

                }
                fprintf(fp,"    };");
                PrCRLF(fp);
        }
}


void PSprite(struct BitMap *bm, FILE *fp, UBYTE *name, int p, BOOL dohead)
{
        UWORD   *wp0, *wp1;     /* Pointer temporaries  */
        short   i, j, nwords,   /* Counter temporaries  */
                color;          /* pixel color          */
        short   wplen = bm->BytesPerRow/2;

        nwords =  2*bm->Rows + (dohead?4:0);
        wp0 = (UWORD *)bm->Planes[p];
        wp1 = (UWORD *)bm->Planes[p+1];

        fprintf(fp, "UWORD %s[%ld] = {", name, nwords);
        PrCRLF(fp);

        if (dohead) {
                fprintf(fp,"  0x0000, 0x0000, /* VStart, VStop */");
                PrCRLF(fp);
        }
        for (j=0 ; j < bm->Rows; j++) {
                fprintf(fp, "  0x%04x, 0x%04x", *wp0, *wp1);
                if (dohead || (j != bm->Rows-1)) {
                        fprintf(fp, ",");
                }
                fprintf(fp, "\t/*  ");
                for (i = 0; i < 16; i++) {
                        color = ((*wp1 >> (14-i)) & 2) + ((*wp0 >> (15-i)) & 1);
                        fprintf(fp, "%c", sp_colors[color]);
                }
                fprintf(fp,"  */");
                PrCRLF(fp);
                wp0 += wplen;
                wp1 += wplen;
        }
        if (dohead)
                fprintf(fp, "  0x0000, 0x0000 }; /* End of Sprite */");
        else
                fprintf(fp," };");
        PrCRLF(fp);
        PrCRLF(fp);
}

void PrintSprite(struct BitMap *bm, FILE *fp, UBYTE *name,
                 BOOL attach, BOOL dohdr)
{
        fprintf(fp,"/*----- Sprite format: h = %ld ------ */", bm->Rows);
        PrCRLF(fp);

        if (bm->Depth > 1) {
                fprintf(fp, "/*--Sprite containing lower order two planes:   */");
                PrCRLF(fp);
                PSprite(bm, fp, name, 0, dohdr);
        }
        if (attach && (bm->Depth > 3) ) {
                strcat(name, "1");
                fprintf(fp, "/*--Sprite containing higher order two planes:   */");

                PrCRLF(fp);
                PSprite(bm, fp, name, 2, dohdr);
        }
}

#define BOB     0
#define SPRITE  1

/* BMPrintCRep
 * Passed pointer to BitMap structure, C filehandle opened for write,
 * name associated with file, and string describing the output
 * format desired (see cases below), outputs C representation of the ILBM.
 */
void BMPrintCRep(struct BitMap *bm, FILE *fp, UBYTE *name, UBYTE *fmt)
{
        BOOL attach, doHdr;
        char c;
        SHORT type;

        doCRLF = NO;
        doHdr = YES;
        type = BOB;
        attach = NO;
        while ( (c=*fmt++) != 0 )
                switch (c) {
                        case 'b':
                                type = BOB;
                                break;
                        case 's':
                                type = SPRITE;
                                attach = NO;
                                break;
                        case 'a':
                                type = SPRITE;
                                attach = YES;
                                break;
                        case 'n':
                                doHdr = NO;
                                break;
                        case 'c':
                                doCRLF = YES;
                                break;
                }
        switch(type) {
                case BOB:
                        PrintBob(bm, fp, name);
                        break;
                case SPRITE:
                        PrintSprite(bm, fp, name, attach, doHdr);
                        break;
        }
}
```

```
/* copychunks
 *
 * For Read/Modify/Write programs and other programs that need
 *   to close the IFF file but still reference gathered chunks.
 * Copies your gathered property and collection chunks
 *   from an iff context so that IFF handle may be
 *   closed right after parsing (allowing file or clipboard to
 *   to be reopened for read or write by self or other programs)
 *
 * The created list of chunks can be modified and written
 *   back out to a new handle with writechunklist().
 *
 * If you have used copychunks(), remember to free the copied
 *   chunks with freechunklist(), when ready, to deallocate them.
 *
 * Note that this implementation is flat and is suitable only
 *   for simple FORMs.
 */

#include "iffp/iff.h"

/* copychunks()
 *
 * Copies chunks specified in propchks and collectchks
 *   FROM an already-parsed IFFHandle
 *   TO a singly linked list of Chunk structures,
 * and returns a pointer to the start of the list.
 *
 * Generally you would store this pointer in parseInfo.copiedchunks.
 *
 * You must later free the list of copied chunks by calling
 *   FreeChunkList().
 *
 * Reorders collection chunks so they appear in SAME ORDER
 * in chunk list as they did in the file.
 *
 * Returns 0 for failure
 */
struct Chunk *copychunks(struct IFFHandle *iff,
                         LONG *propchks, LONG *collectchks,
                         ULONG memtype)
    {
    struct Chunk *chunk, *first=NULL, *prevchunk = NULL;
    struct StoredProperty *sp;
    struct CollectionItem *ci, *cii;
    long error;
    int k, kk, bk;

    if(!iff)    return(NULL);

    /* Copy gathered property chunks */
    error = 0;
    for(k=0; (!error) && (propchks) && (propchks[k] != TAG_DONE); k+=2)
        {
        if(sp=FindProp(iff,propchks[k],propchks[k+1]))
            {
            D(bug("copying %.4s.%.4s chunk\n",&propchks[k],&propchks[k+1]));

            if(chunk=(struct Chunk *)
                     AllocMem(sizeof(struct Chunk),memtype|MEMF_CLEAR))
                {
                chunk->ch_Type = propchks[k];
                chunk->ch_ID   = propchks[k+1];
                if(chunk->ch_Data = AllocMem(sp->sp_Size,memtype))
                    {
                    chunk->ch_Size = sp->sp_Size;
                    CopyMem(sp->sp_Data,chunk->ch_Data,sp->sp_Size);
                    if(prevchunk)        prevchunk->ch_Next = chunk;
                    else                 first = chunk;
                    prevchunk = chunk;
                    }
                else
                    {
                    FreeMem(chunk,sizeof(struct Chunk));
                    chunk=NULL;
                    error = 1;
                    }
                }
            else error = 1;
            }
        }

    /* Copy gathered collection chunks in reverse order */
    for(k=0; (!error) && (collectchks) && (collectchks[k] != TAG_DONE); k+=2)
        {
        if(ci=FindCollection(iff,collectchks[k],collectchks[k+1]))
            {
            D(bug("copying %.4s.%.4s collection\n",&collectchks[k],&collectchks[k+1
]));
            for(cii=ci, bk=0; cii; cii=cii->ci_Next)    bk++;

            D(bug(" There are %ld of these, first is at $%lx\n",bk,ci));

            for( bk; bk; bk--)
                {
                for(kk=1, cii=ci; kk<bk; kk++) cii=cii->ci_Next;

                D(bug("  copying number %ld\n",kk));

                if(chunk=(struct Chunk *)
                    AllocMem(sizeof(struct Chunk),memtype|MEMF_CLEAR))
                    {
                    chunk->ch_Type = collectchks[k];
                    chunk->ch_ID   = collectchks[k+1];
                    if(chunk->ch_Data = AllocMem(cii->ci_Size,memtype))
                        {
                        chunk->ch_Size = cii->ci_Size;
                        CopyMem(cii->ci_Data,chunk->ch_Data,cii->ci_Size);
                        if(prevchunk)    prevchunk->ch_Next = chunk;
                        else             first = chunk;
                        prevchunk = chunk;
                        }
                    else
                        {
                        FreeMem(chunk,sizeof(struct Chunk));
                        chunk=NULL;
                        error = 1;
                        }
                    }
                else error = 1;
                }
            }
        }

    if(error)
        {
        if(first) freechunklist(first);
        first = NULL;
        }

    return(first);
    }
```

```
/* freechunklist - Free a dynamically allocated Chunk list and
 *   all of its ch_Data.
 *
 * Note - if a chunk's ch_Size is IFFSIZE_UNKNOWN, its ch_Data
 *   will not be deallocated.
 */
void freechunklist(struct Chunk *first)
    {
    struct Chunk *chunk, *next;

    chunk = first;
    while(chunk)
        {
        next = chunk->ch_Next;
        if((chunk->ch_Data)&&(chunk->ch_Size != IFFSIZE_UNKNOWN))
                FreeMem(chunk->ch_Data,chunk->ch_Size);
        FreeMem(chunk, sizeof(struct Chunk));
        chunk = next;
        }
    }


/* findchunk - find first matching chunk in list of struct Chunks
 *    example  finchunk(pi->copiedchunks,ID_ILBM,ID_CRNG);
 *
 * returns struct Chunk *, or NULL if none found
 */
struct Chunk *findchunk(struct Chunk *first, long type, long id)
    {
    struct Chunk *chunk;

    for(chunk=first; chunk; chunk=chunk->ch_Next)
        {
        if((chunk->ch_Type == type)&&(chunk->ch_ID == id)) return(chunk);
        }
    return(NULL);
    }


/* writechunklist - write out list of struct Chunk's
 * If data is a null terminated string, you may use
 * IFFSIZE_UNKNOWN as the ch_Szie and strlen(chunk->ch_Data)
 * will be used here as size.
 *
 * Returns 0 for success or an IFFERR
 */
long writechunklist(struct IFFHandle *iff, struct Chunk *first)
    {
    struct Chunk *chunk;
    long size, error = 0;

    D(bug("writechunklist: first chunk pointer = $%lx\n",first));

    for(chunk=first; chunk && (!error); chunk=chunk->ch_Next)
        {
        size  = (chunk->ch_Size == IFFSIZE_UNKNOWN) ?
                        strlen(chunk->ch_Data)  :  chunk->ch_Size;
        error = PutCk(iff, chunk->ch_ID, size, chunk->ch_Data);
        D(bug("writechunklist: put %.4s size=%ld, error=%ld\n",
                                &chunk->ch_ID,size, error));
        }
    return(error);
    }
```

modules/getbitmap.c Page 1

```
/*----------------------------------------------------------------------*
 * GETBITMAP.C  Support routines for reading ILBM files.
 * (IFF is Interchange Format File.)
 *
 * Based on code by Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 * Modified for iffparse.library by CBM 04/90
 * This version for the Commodore-Amiga computer.
 *----------------------------------------------------------------------*/

#include "iffp/ilbm.h"
#include "iffp/packer.h"
#include "iffp/ilbmapp.h"

/* createbrush
 *
 * Passed an initialized ILBMInfo with a parsed IFFHandle (chunks parsed,
 * stopped at BODY),
 * gets the bitmap and colors
 * Sets up ilbm->brbitmap, ilbm->colortable, ilbm->ncolors
 * Returns 0 for success
 */
LONG createbrush(struct ILBMInfo *ilbm)
        {
        int error;

        error                   = getbitmap(ilbm);
        if(!error) error        = loadbody(ilbm->ParseInfo.iff,
                                        ilbm->brbitmap,&ilbm->Bmhd);
        if(!error)              getcolors(ilbm);
        if(error)               deletebrush(ilbm);
        return(error);
        }

/* deletebrush
 *
 * closes and deallocates created brush bitmap and colors
 */
void deletebrush(ilbm)
struct ILBMInfo         *ilbm;
        {
        freebitmap(ilbm);
        freecolors(ilbm);
        }


/* getbitmap
 *
 * Passed an initialized ILBMInfo with parsed IFFHandle (chunks parsed,
 * stopped at BODY), allocates a BitMap structure and planes just large
 * enough for the BODY.  Generally used for brushes but may be used
 * to load backgrounds without displaying, or to load deep ILBM's.
 * Sets ilbm->brbitmap.  Returns 0 for success.
 */
LONG getbitmap(struct ILBMInfo *ilbm)
        {
        struct IFFHandle        *iff;
        BitMapHeader    *bmhd;
        USHORT                  wide, high;
        LONG  error = NULL;
        int k, extra=0;
        BYTE deep;

        if(!(iff=ilbm->ParseInfo.iff))  return(CLIENT_ERROR);
```

modules/getbitmap.c Page 2

```
        ilbm->brbitmap = NULL;

        if (!( bmhd = (BitMapHeader *)
                        findpropdata(iff, ID_ILBM, ID_BMHD)))
                {
                message("No ILBM.BMHD chunk!\n");
                return(IFFERR_SYNTAX);
                }

        *(&ilbm->Bmhd) = *bmhd; /* copy contents of BMHD */

        wide = BitsPerRow(bmhd->w);
        high = bmhd->h;
        deep = bmhd->nPlanes;

        ilbm->camg = getcamg(ilbm);

        D(bug("allocbitmap: bmhd=$%lx wide=%ld high=%ld deep=%ld\n",
                        bmhd,wide,high,deep));
        /*
         * Allocate Bitmap and planes
         */
        extra = deep > 8 ? deep - 8 : 0;
        if(ilbm->brbitmap = AllocMem(sizeof(struct BitMap)+(extra<<2),MEMF_CLEAR))
                {
                InitBitMap(ilbm->brbitmap,deep,wide,high);
                for(k=0; k<deep && (!error); k++)
                    {
                    if(!(ilbm->brbitmap->Planes[k] = AllocRaster(wide,high)))
                        error = 1;
                    if(! error)
                        BltClear(ilbm->brbitmap->Planes[k],RASSIZE(wide,high),0);
                    }

                if(error)
                    {
                    message("Failed to allocate raster\n");
                    freebitmap(ilbm);
                    }
                }
        else error = 1;
        return(error);
        }


/* freebitmap
 *
 * deallocates ilbm->brbitmap BitMap structure and planes
 */
void freebitmap(struct ILBMInfo * ilbm)
        {
        int k, extra=0;

        if(ilbm->brbitmap)
                {
                for(k=0; k< ilbm->brbitmap->Depth; k++)
                        {
                        if(ilbm->brbitmap->Planes[k])
                                FreeRaster(ilbm->brbitmap->Planes[k],
                                        (USHORT)(ilbm->brbitmap->BytesPerRow << 3),
                                        ilbm->brbitmap->Rows);
                        }

                extra = ilbm->brbitmap->Depth > 8 ? ilbm->brbitmap->Depth - 8 : 0;
                FreeMem(ilbm->brbitmap,sizeof(struct BitMap) + (extra << 2));
                ilbm->brbitmap = NULL;
```

```
                        }
                }

/* end */
```

```
/*----------------------------------------------------------------------*
 * GETDISPLAY.C  Support routines for reading/displaying ILBM files.
 * (IFF is Interchange Format File.)
 *
 * Based on code by Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 * Modified for iffparse.library by CBM 04/90
 * This version for the Commodore-Amiga computer.
 *----------------------------------------------------------------------*/

#include "iffp/ilbm.h"
#include "iffp/packer.h"
#include "iffp/ilbmapp.h"

extern struct Library *GfxBase;

/* showilbm
 *
 * Passed an ILBMInfo initilized with with a not-in-use ParseInfo.iff
 *   IFFHandle and desired propchks, collectchks, stopchks, and a filename,
 *   will load and display an ILBM, initializing ilbm->Bmhd, ilbm->camg,
 *   ilbm->scr, ilbm->win, ilbm->vp, ilbm->srp, ilbm->wrp,
 *   ilbm->colortable, and ilbm->ncolors.
 *
 *   Note that ncolors may be more colors than you can LoadRGB4.
 *   Use MIN(ilbm->ncolors,MAXAMCOLORREG) for color count if you change
 *   the colors yourself using 1.3/2.0 functions.
 *
 * Returns 0 for success or an IFFERR (libraries/iffparse.h)
 */

LONG showilbm(struct ILBMInfo *ilbm, UBYTE *filename)
{
LONG error = 0L;

    if(!(ilbm->ParseInfo.iff))  return(CLIENT_ERROR);

    if(!(error = openifile((struct ParseInfo *)ilbm, filename, IFFF_READ)))
        {
        D(bug("showilbm: openifile successful\n"));

        error = parseifile((struct ParseInfo *)ilbm,
                                ID_FORM, ID_ILBM,
                                ilbm->ParseInfo.propchks,
                                ilbm->ParseInfo.collectchks,
                                ilbm->ParseInfo.stopchks);

        if((!error)||(error == IFFERR_EOC)||(error == IFFERR_EOF))
            {
            D(bug("showilbm: parseifile successful\n"));

            if(contextis(ilbm->ParseInfo.iff,ID_ILBM,ID_FORM))
                {
                if(error = createdisplay(ilbm)) deletedisplay(ilbm);
                }
            else
                {
                closeifile((struct ParseInfo *)ilbm);
                message("Not an ILBM\n");
                error = NOFILE;
                }
            }
        }
    return(error);
}
```

```
/* unshowilbm
 *
 * frees and closes everything alloc'd/opened by showilbm
 */
void unshowilbm(struct ILBMInfo *ilbm)
{
        deletedisplay(ilbm);
        closeifile((struct ParseInfo *)ilbm);
}


/* createdisplay
 *
 * Passed a initialized ILBMInfo with parsed IFFHandle (chunks parsed,
 * stopped at BODY),
 * opens/allocs the display and colortable, and displays the ILBM.
 *
 * If successful, sets up ilbm->Bmhd, ilbm->camg, ilbm->scr, ilbm->win,
 *   ilbm->vp,  ilbm->wrp, ilbm->srp and also ilbm->colortable and
 *   ilbm->ncolors.
 *
 * Note that ncolors may be more colors than you can LoadRGB4.
 * Use MIN(ilbm->ncolors,MAXAMCOLORREG) for color count if you change
 *   the colors yourself using 1.3/2.0 functions.
 *
 * Returns 0 for success or an IFFERR (libraries/iffparse.h)
 */

LONG createdisplay(struct ILBMInfo *ilbm)
        {
        int error;

        D(bug("createdisplay:\n"));

        error                   = getdisplay(ilbm);

        D(bug("createdisplay: after getdisplay, error = %ld\n", error));

        if(!error)      error   = loadbody(ilbm->ParseInfo.iff,
                                        &ilbm->scr->BitMap,&ilbm->Bmhd);

        D(bug("createdisplay: after loadbody, error = %ld\n", error));

        if(!error)
                {
                if(!(getcolors(ilbm)))
                   LoadRGB4(ilbm->vp, ilbm->colortable,
                                MIN(ilbm->ncolors,MAXAMCOLORREG));
                }
        if(error)  deletedisplay(ilbm);
        return(error);
        }

/* deletedisplay
 *
 * closes and deallocates created display and colors
 */
void deletedisplay(struct ILBMInfo *ilbm)
        {
        freedisplay(ilbm);
        freecolors(ilbm);
        }
```

```
/* getdisplay
 *
 * Passed an initialized ILBMInfo with a parsed IFFHandle (chunks parsed,
 * stopped at BODY),
 * gets the dimensions and mode for the display and calls the external
 * routine opendisplay().  Our opendisplay() is in the screen.c
 * module.  It opens a 2.0 or 1.3, ECS or non-ECS screen and window.
 * It also does 2.0 overscan centering based on the closest user prefs.
 *
 * If successful, sets up ilbm->Bmhd, ilbm->camg, ilbm->scr, ilbm->win,
 *   ilbm->vp, ilbm->wrp, ilbm->srp
 *
 * Returns 0 for success or an IFFERR (libraries/iffparse.h)
 */
LONG getdisplay(struct ILBMInfo *ilbm)
        {
        struct IFFHandle *iff;
        BitMapHeader *bmhd;
        ULONG                           modeid;
        UWORD                           wide, high, deep;


        if(!(iff=ilbm->ParseInfo.iff))  return(CLIENT_ERROR);

        if(!(bmhd = (BitMapHeader *)findpropdata(iff, ID_ILBM, ID_BMHD)))
                {
                message ("No ILBM.BMHD chunk\n");
                return(IFFERR_SYNTAX);
                }

        *(&ilbm->Bmhd)  = *bmhd;

        wide = (RowBytes(bmhd->w)) >= (RowBytes(bmhd->pageWidth)) ?
                bmhd->w : bmhd->pageWidth;
        high = MAX(bmhd->h, bmhd->pageHeight);
        deep = MIN(bmhd->nPlanes,MAXAMDEPTH);

        ilbm->camg = modeid = getcamg(ilbm);

        /*
         * Open the display
         */
        if(!(opendisplay(ilbm,wide,high,deep,modeid)))
                {
                message("Failed to open display.\n");
                return(1);
                }
        return(0);
        }


/* freedisplay
 *
 * closes and deallocates display from getdisplay (not colors)
 */
void freedisplay(struct ILBMInfo *ilbm)
        {
        closedisplay(ilbm);
        }
```

```
/* ilbmr.c --- ILBM loading routines for use with iffparse */

/*----------------------------------------------------------------------*
 * ILBMR.C  Support routines for reading ILBM files.
 * (IFF is Interchange Format File.)
 *
 * Based on code by Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 * Modified for iffparse.library 05/90
 * This version for the Commodore-Amiga computer.
 *----------------------------------------------------------------------*/

#include "iffp/ilbm.h"
#include "iffp/packer.h"
#include "iffp/ilbmapp.h"

#define movmem CopyMem

#define MaxSrcPlanes (25)

extern struct Library *GfxBase;

/*---------- loadbody ---------------------------------------------------*/

LONG loadbody(iff, bitmap, bmhd)
struct IFFHandle *iff;
struct BitMap *bitmap;
BitMapHeader *bmhd;
        {
        BYTE *buffer;
        ULONG bufsize;
        LONG error = 1;

        D(bug("In loadbody\n"));

        if(!(currentchunkis(iff,ID_ILBM,ID_BODY)))
            {
            message("ILBM has no BODY\n");      /* Maybe it's a palette */
            return(IFF_OKAY);
            }

        if((bitmap)&&(bmhd))
            {
            D(bug("Have bitmap and bmhd\n"));

            bufsize = MaxPackedSize(RowBytes(bmhd->w)) << 4;
            if(!(buffer = AllocMem(bufsize,0L)))
                {
                D(bug("Buffer alloc of %ld failed\n",bufsize));
                return(IFFERR_NOMEM);
                }
            error = loadbody2(iff, bitmap, NULL, bmhd, buffer, bufsize);
            D(bug("Returned from getbody, error = %ld\n",error));
            }
        FreeMem(buffer,bufsize);
        return(error);
        }


/* like the old GetBODY */
LONG loadbody2(iff, bitmap, mask, bmhd, buffer, bufsize)
struct IFFHandle *iff;
struct BitMap *bitmap;
BYTE *mask;
BitMapHeader *bmhd;
BYTE *buffer;
```

```
ULONG bufsize;
   {
   UBYTE srcPlaneCnt = bmhd->nPlanes;   /* Haven't counted for mask plane yet*/
   WORD srcRowBytes = RowBytes(bmhd->w);
   WORD destRowBytes = bitmap->BytesPerRow;
   LONG bufRowBytes = MaxPackedSize(srcRowBytes);
   int nRows = bmhd->h;
   WORD compression = bmhd->compression;
   register int iPlane, iRow, nEmpty;
   register WORD nFilled;
   BYTE *buf, *nullDest, *nullBuf, **pDest;
   BYTE *planes[MaxSrcPlanes]; /* array of ptrs to planes & mask */
   struct ContextNode *cn;

   D(bug("srcRowBytes = %ld\n",srcRowBytes));

   cn = CurrentChunk(iff);

   if (compression > cmpByteRun1)
      return(CLIENT_ERROR);

   D(bug("loadbody2: compression=%ld srcBytes=%ld bitmapBytes=%ld\n",
                compression, srcRowBytes, bitmap->BytesPerRow));
   D(bug("loadbody2: bufsize=%ld bufRowBytes=%ld, srcPlaneCnt=%ld\n",
                        bufsize, bufRowBytes, srcPlaneCnt));

   /* Complain if client asked for a conversion GetBODY doesn't handle.*/
   if ( srcRowBytes  >  bitmap->BytesPerRow  ||
         bufsize < bufRowBytes * 2  ||
         srcPlaneCnt > MaxSrcPlanes )
      return(CLIENT_ERROR);

   D(bug("loadbody2: past conversion checks\n"));

   if (nRows > bitmap->Rows)   nRows = bitmap->Rows;

   D(bug("loadbody2: srcRowBytes=%ld, srcRows=%ld, srcDepth=%ld, destDepth=%ld\n",
                srcRowBytes, nRows, bmhd->nPlanes, bitmap->Depth));

   /* Initialize array "planes" with bitmap ptrs; NULL in empty slots.*/
   for (iPlane = 0; iPlane < bitmap->Depth; iPlane++)
      planes[iPlane] = (BYTE *)bitmap->Planes[iPlane];
   for ( ;  iPlane < MaxSrcPlanes;  iPlane++)
      planes[iPlane] = NULL;

   /* Copy any mask plane ptr into corresponding "planes" slot.*/
   if (bmhd->masking == mskHasMask)
        {
        if (mask != NULL)
             planes[srcPlaneCnt] = mask;  /* If there are more srcPlanes than
                * dstPlanes, there will be NULL plane-pointers before this.*/
        else
             planes[srcPlaneCnt] = NULL;  /* In case more dstPlanes than src.*/
        srcPlaneCnt += 1;  /* Include mask plane in count.*/
        }

   /* Setup a sink for dummy destination of rows from unwanted planes.*/
   nullDest = buffer;
   buffer  += srcRowBytes;
   bufsize -= srcRowBytes;

   /* Read the BODY contents into client's bitmap.
    * De-interleave planes and decompress rows.
    * MODIFIES: Last iteration modifies bufsize.*/

   buf = buffer + bufsize;  /* Buffer is currently empty.*/
```

```
    for (iRow = nRows; iRow > 0; iRow--)
          {
          for (iPlane = 0; iPlane < srcPlaneCnt; iPlane++)
              {
              pDest = &planes[iPlane];

              /* Establish a sink for any unwanted plane.*/
              if (*pDest == NULL)
                  {
                  nullBuf = nullDest;
                  pDest   = &nullBuf;
                  }

              /* Read in at least enough bytes to uncompress next row.*/
              nEmpty  = buf - buffer;       /* size of empty part of buffer.*/
              nFilled = bufsize - nEmpty;   /* this part has data.*/
              if (nFilled < bufRowBytes)
                  {
                  /* Need to read more.*/

                  /* Move the existing data to the front of the buffer.*/
                  /* Now covers range buffer[0]..buffer[nFilled-1].*/
                  movmem(buf, buffer, nFilled);  /* Could be moving 0 bytes.*/

                  if(nEmpty > ChunkMoreBytes(cn))
                      {
                      /* There aren't enough bytes left to fill the buffer.*/
                      nEmpty = ChunkMoreBytes(cn);
                      bufsize = nFilled + nEmpty;  /* heh-heh */
                      }

                  /* Append new data to the existing data.*/
                  if(ReadChunkBytes(iff, &buffer[nFilled], nEmpty) < nEmpty)
                          return(CLIENT_ERROR);

                  buf     = buffer;
                  nFilled = bufsize;
                  nEmpty  = 0;
                  }

              /* Copy uncompressed row to destination plane.*/
              if(compression == cmpNone)
                  {
                  if(nFilled < srcRowBytes)  return(IFFERR_MANGLED);
                  movmem(buf, *pDest, srcRowBytes);
                  buf    += srcRowBytes;
                  *pDest += destRowBytes;
                  }
              else
                  {
                  /* Decompress row to destination plane.*/
                  if ( unpackrow(&buf, pDest, nFilled,  srcRowBytes) )
                      /*  pSource, pDest, srcBytes, dstBytes  */
                          return(IFFERR_MANGLED);
                  else *pDest += (destRowBytes - srcRowBytes);
                  }
              }
          }
    return(IFF_OKAY);
    }


/* ----------- getcolors ------------- */

/* getcolors - allocates a ilbm->colortable for at least MAXAMCOLORREG
 *      and loads CMAP colors into it, setting ilbm->ncolors to number
```

```
 *      of colors actually loaded.
 */
LONG getcolors(struct ILBMInfo *ilbm)
        {
        struct IFFHandle        *iff;
        int error = 1;

        if(!(iff=ilbm->ParseInfo.iff))  return(CLIENT_ERROR);

        if(!(error = alloccolortable(ilbm)))
           error = loadcmap(iff, ilbm->colortable, &ilbm->ncolors);
        if(error) freecolors(ilbm);
        D(bug("getcolors: error = %ld\n",error));
        return(error);
        }


/* alloccolortable - allocates ilbm->colortable and sets ilbm->ncolors
 *      to the number of colors we have room for in the table.
 */

LONG alloccolortable(struct ILBMInfo *ilbm)
        {
        struct IFFHandle        *iff;
        struct  StoredProperty  *sp;

        LONG    error = CLIENT_ERROR;
        ULONG   ctabsize;
        USHORT  ncolors;

        if(!(iff=ilbm->ParseInfo.iff))  return(CLIENT_ERROR);

        if(sp = FindProp (iff, ID_ILBM, ID_CMAP))
                {
                /*
                 * Compute the size table we need
                 */
                ncolors = sp->sp_Size / 3;              /* how many in CMAP */
                ncolors = MAX(ncolors, MAXAMCOLORREG);

                ctabsize = ncolors * sizeof(Color4);
                if(ilbm->colortable =
                   (Color4 *)AllocMem(ctabsize,MEMF_CLEAR|MEMF_PUBLIC))
                    {
                    ilbm->ncolors = ncolors;
                    ilbm->ctabsize = ctabsize;
                    error = 0L;
                    }
                else error = IFFERR_NOMEM;
                }
        D(bug("alloccolortable for %ld colors: error = %ld\n",ncolors,error));
        return(error);
        }


void freecolors(struct ILBMInfo *ilbm)
        {
        if(ilbm->colortable)
                {
                FreeMem(ilbm->colortable, ilbm->ctabsize);
                }
        ilbm->colortable = NULL;
        ilbm->ctabsize = 0;
        }
```

```
/* Passed IFFHandle, pointer to colortable array, and pointer to
 * a USHORT containing number of colors caller has space to hold,
 * loads the colors and sets pNcolors to the number actually read.
 *
 * NOTE !!! - Old GetCMAP passed a pointer to a UBYTE for pNcolors
 *            This one is passed a pointer to a USHORT
 */
LONG loadcmap(struct IFFHandle *iff, WORD *colortable,USHORT *pNcolors)
        {
        register struct StoredProperty  *sp;
        register LONG                   idx;
        register ULONG                  ncolors;
        register UBYTE                  *rgb;
        LONG                            r, g, b;

        if(!(colortable))
                {
                message("No colortable allocated\n");
                return(1);
                }

        if(!(sp = FindProp (iff, ID_ILBM, ID_CMAP)))    return(1);

        rgb = sp->sp_Data;
        ncolors = sp->sp_Size / sizeofColorRegister;
        if(*pNcolors < ncolors) ncolors = *pNcolors;
        *pNcolors = ncolors;

        idx = 0;
        while (ncolors--)
                {
                r = (*rgb++ & 0xF0) << 4;
                g = *rgb++ & 0xF0;
                b = *rgb++ >> 4;
                colortable[idx] = r | g | b;
                idx++;
                }
        return(0);
        }

/*
 * Returns CAMG or computed mode for storage in ilbm->camg
 *
 * ilbm->Bmhd structure must be initialized prior to this call.
 */
ULONG getcamg(struct ILBMInfo *ilbm)
        {
        struct IFFHandle *iff;
        struct StoredProperty *sp;
        UWORD  wide,high,deep;
        ULONG modeid = 0L;

        if(!(iff=ilbm->ParseInfo.iff))  return(0L);

        wide = ilbm->Bmhd.pageWidth;
        high = ilbm->Bmhd.pageHeight;
        deep = ilbm->Bmhd.nPlanes;

        D(bug("Getting CAMG for w=%ld h=%ld d=%ld ILBM\n",wide,high,deep));

        /*
         * Grab CAMG's idea of the viewmodes.
         */
        if (sp = FindProp (iff, ID_ILBM, ID_CAMG))
                {
```

```
                modeid = (* (ULONG *) sp->sp_Data);

                /* knock bad bits out of old-style 16-bit viewmode CAMGs
                 */
                if((!(modeid & MONITOR_ID_MASK))||
                  ((modeid & EXTENDED_MODE)&&(!(modeid & 0xFFFF0000))))
                   modeid &=
                     (~(EXTENDED_MODE|SPRITES|GENLOCK_AUDIO|GENLOCK_VIDEO|VP_HIDE));

                /* check for bogus CAMG like DPaintII brushes
                 * with junk in upper word and extended bit
                 * not set in lower word.
                 */
                if((modeid & 0xFFFF0000)&&(!(modeid & 0x00001000))) sp=NULL;
                }

        if(!sp) {
                /*
                 * No CAMG (or bad CAMG) present; use computed modes.
                 */
                if (wide >= 640)        modeid = HIRES;
                if (high >= 400)        modeid |= LACE;
                if (deep == 6)
                        {
                        modeid |= ilbm->EHB ? EXTRA_HALFBRITE : HAM;
                        }
                D(bug("No CAMG found - using mode $%08lx\n",modeid));
                }

        D(bug("getcamg: modeid = $%08lx\n",modeid));
        return(modeid);
        }
```

modules/ilbmw.c Page 1

```
/*----------------------------------------------------------------------*
 * ILBMW.C  Support routines for writing ILBM files using IFFParse.
 * (IFF is Interchange Format File.)
 *
 * Based on code by Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 * This version for the Commodore-Amiga computer.
 *----------------------------------------------------------------------*/

#include "iffp/ilbm.h"
#include "iffp/packer.h"

#include <graphics/gfxbase.h>

extern struct Library *GfxBase;

/*---------- initbmhd -------------------------------------------------*/
long initbmhd(BitMapHeader *bmhd, struct BitMap *bitmap,
              WORD masking, WORD compression, WORD transparentColor,
              WORD width, WORD height, WORD pageWidth, WORD pageHeight,
              ULONG modeid)
    {
    extern struct Library *GfxBase;
    struct DisplayInfo DI;

    WORD rowBytes = bitmap->BytesPerRow;

    D(bug("In InitBMHD\n"));

    bmhd->w = width;
    bmhd->h = height;
    bmhd->x = bmhd->y = 0;      /* Default position is (0,0).*/
    bmhd->nPlanes = bitmap->Depth;
    bmhd->masking = masking;
    bmhd->compression = compression;
    bmhd->reserved1 = 0;
    bmhd->transparentColor = transparentColor;
    bmhd->pageWidth = pageWidth;
    bmhd->pageHeight = pageHeight;

    bmhd->xAspect = 0;  /* So we can tell when we've got it */
    if(GfxBase->lib_Version >=36)
        {
        if(GetDisplayInfoData(NULL, (UBYTE *)&DI,
                sizeof(struct DisplayInfo), DTAG_DISP, modeid))
                {
                bmhd->xAspect =  DI.Resolution.x;
                bmhd->yAspect =  DI.Resolution.y;
                }
        }

    /* If running under 1.3 or GetDisplayInfoData failed, use old method
     * of guessing aspect ratio
     */
    if(! bmhd->xAspect)
        {
        bmhd->xAspect =  44;
        bmhd->yAspect =
                ((struct GfxBase *)GfxBase)->DisplayFlags & PAL ? 44 : 52;
        if(modeid & HIRES)      bmhd->xAspect = bmhd->xAspect >> 1;
        if(modeid & LACE)       bmhd->yAspect = bmhd->yAspect >> 1;
        }

    return( IS_ODD(rowBytes) ? CLIENT_ERROR : IFF_OKAY );
    }
```

modules/ilbmw.c Page 2

```
/*---------- putcmap ---------------------------------------------------*/
/* This function will accept a table of color values in one of the
 * following forms:
 *  if bitspergun=4,  colortable is words, each with nibbles 0RGB
 *  if bitspergun=8,  colortable is bytes of RGBRGB etc. (like a CMAP)
 *  if bitspergun=32, colortable is ULONGS of RGBRGB etc.
 *     (only the high eight bits of each gun will be written to CMAP)
 */
long putcmap(struct IFFHandle *iff, APTR colortable,
             UWORD ncolors, UWORD bitspergun)
    {
    long error, offs;
    WORD  *tabw;
    UBYTE *tab8;
    ColorRegister cmapReg;

    D(bug("In PutCMAP\n"));

    if((!iff)||(!colortable))    return(CLIENT_ERROR);

    /* size of CMAP is 3 bytes * ncolors */
    if(error = PushChunk(iff, NULL, ID_CMAP, ncolors * sizeofColorRegister))
        return(error);

    D(bug("Pushed ID_CMAP, error = %ld\n",error));

    if(bitspergun == 4)
        {
        /* Store each 4-bit value n as nn */
        tabw = (UWORD *)colortable;
        for( ;  ncolors;  --ncolors )
            {
            cmapReg.red    = ( *tabw >> 4 ) & 0xf0;
            cmapReg.red   |= (cmapReg.red >> 4);

            cmapReg.green  = ( *tabw      ) & 0xf0;
            cmapReg.green |= (cmapReg.green >> 4);

            cmapReg.blue   = ( *tabw << 4 ) & 0xf0;
            cmapReg.blue  |= (cmapReg.blue >> 4);

            if((WriteChunkBytes(iff, (BYTE *)&cmapReg, sizeofColorRegister))
                != sizeofColorRegister)
                    return(IFFERR_WRITE);
            ++tabw;
            }
        }
    else if((bitspergun == 8)||(bitspergun == 32))
        {
        tab8 = (UBYTE *)colortable;
        offs = (bitspergun == 8) ? 1 : 4;
        for( ;  ncolors;  --ncolors )
            {
            cmapReg.red   = *tab8;
            tab8 += offs;
            cmapReg.green = *tab8;
            tab8 += offs;
            cmapReg.blue  = *tab8;
            tab8 += offs;
            if((WriteChunkBytes(iff, (BYTE *)&cmapReg, sizeofColorRegister))
                != sizeofColorRegister)
                    return(IFFERR_WRITE);
            }
        }
    else (error = CLIENT_ERROR)
```

```
   D(bug("Wrote registers, error = %ld\n",error));

   error = PopChunk(iff);
   return(error);
   }

/*---------- putbody -----------------------------------------------------*/
/* NOTE: This implementation could be a LOT faster if it used more of the
 * supplied buffer. It would make far fewer calls to IFFWriteBytes (and
 * therefore to DOS Write). */
long putbody(struct IFFHandle *iff, struct BitMap *bitmap, BYTE *mask,
             BitMapHeader *bmhd, BYTE *buffer, LONG bufsize)
   {
   long error;
   LONG rowBytes = bitmap->BytesPerRow;
   int dstDepth = bmhd->nPlanes;
   UBYTE compression = bmhd->compression;
   int planeCnt;                /* number of bit planes including mask */
   register int iPlane, iRow;
   register LONG packedRowBytes;
   BYTE *buf;
   BYTE *planes[MAXSAVEDEPTH + 1]; /* array of ptrs to planes & mask */

   D(bug("In PutBODY\n"));

   if ( bufsize < MaxPackedSize(rowBytes)  ||   /* Must buffer a comprsd row*/
        compression > cmpByteRun1  ||           /* bad arg */
        bitmap->Rows != bmhd->h   ||            /* inconsistent */
        rowBytes != RowBytes(bmhd->w)  ||       /* inconsistent*/
        bitmap->Depth < dstDepth   ||           /* inconsistent */
        dstDepth > MAXSAVEDEPTH )               /* too many for this routine*/
      return(CLIENT_ERROR);

   planeCnt = dstDepth + (mask == NULL ? 0 : 1);

   /* Copy the ptrs to bit & mask planes into local array "planes" */
   for (iPlane = 0; iPlane < dstDepth; iPlane++)
      planes[iPlane] = (BYTE *)bitmap->Planes[iPlane];
   if (mask != NULL)
      planes[dstDepth] = mask;

   /* Write out a BODY chunk header */
   if(error = PushChunk(iff, NULL, ID_BODY, IFFSIZE_UNKNOWN)) return(error);

   /* Write out the BODY contents */
   for (iRow = bmhd->h; iRow > 0; iRow--)  {
      for (iPlane = 0; iPlane < planeCnt; iPlane++)  {

         /* Write next row.*/
         if (compression == cmpNone) {
            if(WriteChunkBytes(iff,planes[iPlane],rowBytes) != rowBytes)
               error = IFFERR_WRITE;
            planes[iPlane] += rowBytes;
            }

         /* Compress and write next row.*/
         else {
            buf = buffer;
            packedRowBytes = packrow(&planes[iPlane], &buf, rowBytes);
            if(WriteChunkBytes(iff,buffer,packedRowBytes) != packedRowBytes)
               error = IFFERR_WRITE;
            }

         if(error)      return(error);
         }
```

```
      }

   /* Finish the chunk */
   error = PopChunk(iff);
   return(error);
   }
```

modules/loadilbm.c Page 1

```
/* loadilbm.c 05/91  C. Scheppner CBM
 *
 * High-level ILBM load routines
 */

#include "iffp/ilbm.h"
#include "iffp/ilbmapp.h"

extern struct Library *GfxBase;

/* loadbrush
 *
 * Passed an initialized ILBMInfo with a not-in-use ParseInfo.iff
 *   IFFHandle and desired propchks, collectchks, and stopchks, and filename,
 *   will load an ILBM as a brush, setting up ilbm->Bmhd, ilbm->camg,
 *   ilbm->brbitmap, ilbm->colortable, and ilbm->ncolors
 *
 *   Note that ncolors may be more colors than you can LoadRGB4.
 *   Use MIN(ilbm->ncolors,MAXAMCOLORREG) for color count if you change
 *   the colors yourself using 1.3/2.0 functions.
 *
 * Returns 0 for success or an IFFERR (libraries/iffparse.h)
 */

LONG loadbrush(struct ILBMInfo *ilbm, UBYTE *filename)
{
LONG error = 0L;

    if(!(ilbm->ParseInfo.iff))  return(CLIENT_ERROR);

    if(!(error = openifile((struct ParseInfo *)ilbm, filename, IFFF_READ)))
        {
        error = parseifile((struct ParseInfo *)ilbm,
                            ID_FORM, ID_ILBM,
                            ilbm->ParseInfo.propchks,
                            ilbm->ParseInfo.collectchks,
                            ilbm->ParseInfo.stopchks);
        if((!error)||(error == IFFERR_EOC)||(error == IFFERR_EOF))
            {
            if(contextis(ilbm->ParseInfo.iff,ID_ILBM,ID_FORM))
                {
                if(error = createbrush(ilbm))   deletebrush(ilbm);
                }
            else
                {
                closeifile((struct ParseInfo *)ilbm);
                message("Not an ILBM\n");
                error = NOFILE;
                }
            }
        }
    return(error);
}


/* unloadbrush
 *
 * frees and close everything alloc'd/opened by loadbrush
 */
void unloadbrush(struct ILBMInfo *ilbm)
{
    closeifile((struct ParseInfo *)ilbm);
    deletebrush(ilbm);
}
```

modules/loadilbm.c Page 2

```
/* queryilbm
 *
 * Passed an initilized ILBMInfo with a not-in-use IFFHandle,
 *   and a filename,
 *   will open an ILBM, fill in ilbm->camg and ilbm->bmhd,
 *   and close the ILBM.
 *
 * This allows you to determine if the ILBM is a size and
 *   type you want to deal with.
 *
 * Returns 0 for success or an IFFERR (libraries/iffparse.h)
 */

/* query just wants these chunks */
LONG queryprops[] = { ID_ILBM, ID_BMHD,
                      ID_ILBM, ID_CAMG,
                      TAG_DONE };

/* scan can stop when a CMAP or BODY is reached */
LONG querystops[] = { ID_ILBM, ID_CMAP,
                      ID_ILBM, ID_BODY,
                      TAG_DONE };

LONG queryilbm(struct ILBMInfo *ilbm, UBYTE *filename)
{
LONG error = 0L;
BitMapHeader *bmhd;

    if(!(ilbm->ParseInfo.iff))  return(CLIENT_ERROR);

    if(!(error = openifile((struct ParseInfo *)ilbm, filename, IFFF_READ)))
        {
        D(bug("queryilbm: openifile successful\n"));

        error = parseifile((struct ParseInfo *)ilbm,
                        ID_FORM, ID_ILBM,
                        queryprops, NULL, querystops);

        D(bug("queryilbm: after parseifile, error = %ld\n",error));

        if((!error)||(error == IFFERR_EOC)||(error == IFFERR_EOF))
            {
            if(contextis(ilbm->ParseInfo.iff,ID_ILBM,ID_FORM))
                {
                if(bmhd = (BitMapHeader*)
                        findpropdata(ilbm->ParseInfo.iff,ID_ILBM,ID_BMHD))
                    {
                    *(&ilbm->Bmhd) = *bmhd;
                    ilbm->camg = getcamg(ilbm);
                    }
                else error = NOFILE;
                }
            else
                {
                message("Not an ILBM\n");
                error = NOFILE;
                }
            }
        closeifile(ilbm);
        }
    return(error);
}


/* loadilbm
 *
 * Passed a not-in-use IFFHandle, an initialized ILBMInfo, and filename,
 *   will load an ILBM into your already opened ilbm->scr, setting up
 *   ilbm->Bmhd, ilbm->camg, ilbm->colortable, and ilbm->ncolors
 *   and loading the colors into the screen's viewport
 *
 *   Note that ncolors may be more colors than you can LoadRGB4.
 *   Use MIN(ilbm->ncolors,MAXAMCOLORREG) for color count if you change
 *   the colors yourself using 1.3/2.0 functions.
 *
 * Returns 0 for success or an IFFERR (libraries/iffparse.h)
 *
 * NOTE - loadilbm() keeps the IFFHandle open so you can copy
 *   or examine other chunks.  You must call closeifile(iff,ilbm)
 *   to close the file and deallocate the parsed context
 *
 */

LONG loadilbm(struct ILBMInfo *ilbm, UBYTE *filename)
{
LONG error = 0L;


    D(bug("loadilbm:\n"));

    if(!(ilbm->ParseInfo.iff))  return(CLIENT_ERROR);
    if(!ilbm->scr)              return(CLIENT_ERROR);

    if(!(error = openifile((struct ParseInfo *)ilbm, filename, IFFF_READ)))
        {
        D(bug("loadilbm: openifile successful\n"));

        error = parseifile((struct ParseInfo *)ilbm,
                        ID_FORM, ID_ILBM,
                        ilbm->ParseInfo.propchks,
                        ilbm->ParseInfo.collectchks,
                        ilbm->ParseInfo.stopchks);

        D(bug("loadilbm: after parseifile, error = %ld\n",error));

        if((!error)||(error == IFFERR_EOC)||(error == IFFERR_EOF))
            {
            if(contextis(ilbm->ParseInfo.iff,ID_ILBM,ID_FORM))
                {
                error = loadbody(ilbm->ParseInfo.iff,
                                &ilbm->scr->BitMap, &ilbm->Bmhd);

                D(bug("loadilbm: after loadbody, error = %ld\n",error));

                if(!error)
                    {
                    if(!(getcolors(ilbm)))
                        LoadRGB4(&ilbm->scr->ViewPort,ilbm->colortable,
                                MIN(ilbm->ncolors,MAXAMCOLORREG));
                    }
                }
            else
                {
                closeifile((struct ParseInfo *)ilbm);
                message("Not an ILBM\n");
                error = NOFILE;
                }
            }
        }
    return(error);
}
```

```
/* unloadilbm
 *
 * frees and closes everything allocated by loadilbm
 */
void unloadilbm(struct ILBMInfo *ilbm)
{
    closeifile((struct ParseInfo *)ilbm);
    freecolors(ilbm);
}
```

```
/*----------------------------------------------------------------------*
 * packer.c Convert data to "cmpByteRun1" run compression.     11/15/85
 *
 * By Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 *      control bytes:
 *       [0..127]   : followed by n+1 bytes of data.
 *       [-1..-127] : followed by byte to be repeated (-n)+1 times.
 *       -128       : NOOP.
 *
 * This version for the Commodore-Amiga computer.
 *----------------------------------------------------------------------*/
#include "iffp/packer.h"

#define DUMP    0
#define RUN     1

#define MinRun 3
#define MaxRun 128
#define MaxDat 128

/* When used on global definitions, static means private.
 * This keeps these names, which are only referenced in this
 * module, from conficting with same-named objects in your program.
 */
static LONG putSize;
static char buf[256];   /* [TBD] should be 128?  on stack?*/

#define GetByte()       (*source++)
#define PutByte(c)      { *dest++ = (c);   ++putSize; }

static BYTE *PutDump(dest, nn)  BYTE *dest;  int nn; {
        int i;

        PutByte(nn-1);
        for(i = 0;  i < nn;  i++)   PutByte(buf[i]);
        return(dest);
        }

static BYTE *PutRun(dest, nn, cc)   BYTE *dest;  int nn, cc; {
        PutByte(-(nn-1));
        PutByte(cc);
        return(dest);
        }

#define OutDump(nn)   dest = PutDump(dest, nn)
#define OutRun(nn,cc) dest = PutRun(dest, nn, cc)

/*----------- packrow --------------------------------------------------*/
/* Given POINTERS TO POINTERS, packs one row, updating the source and
 * destination pointers.  RETURNs count of packed bytes.
 */
LONG packrow(BYTE **pSource, BYTE **pDest, LONG rowSize)
    {
    BYTE *source, *dest;
    char c,lastc = '\0';
    BOOL mode = DUMP;
    short nbuf = 0;             /* number of chars in buffer */
    short rstart = 0;           /* buffer index current run starts */

    source = *pSource;
    dest = *pDest;
    putSize = 0;
    buf[0] = lastc = c = GetByte();  /* so have valid lastc */
    nbuf = 1;   rowSize--;      /* since one byte eaten.*/
```

```
    for (;  rowSize;  --rowSize) {
        buf[nbuf++] = c = GetByte();
        switch (mode) {
                case DUMP:
                        /* If the buffer is full, write the length byte,
                           then the data */
                        if (nbuf>MaxDat) {
                                OutDump(nbuf-1);
                                buf[0] = c;
                                nbuf = 1;   rstart = 0;
                                break;
                                }

                        if (c == lastc) {
                            if (nbuf-rstart >= MinRun) {
                                if (rstart > 0) OutDump(rstart);
                                mode = RUN;
                                }
                            else if (rstart == 0)
                                mode = RUN;     /* no dump in progress,
                                so can't lose by making these 2 a run.*/
                            }
                        else  rstart = nbuf-1;          /* first of run */
                        break;

                case RUN: if ( (c != lastc)|| ( nbuf-rstart > MaxRun)) {
                        /* output run */
                        OutRun(nbuf-1-rstart,lastc);
                        buf[0] = c;
                        nbuf = 1; rstart = 0;
                        mode = DUMP;
                        }
                        break;
                }

        lastc = c;
        }

    switch (mode) {
        case DUMP: OutDump(nbuf); break;
        case RUN: OutRun(nbuf-rstart,lastc); break;
        }
    *pSource = source;
    *pDest = dest;
    return(putSize);
    }
```

```
/*
 * parse.c - iffparse file IO support module
 *   based on some of looki.c by Leo Schwab
 *
 * The filename for clipboard is -c or -cUnit as in -c0 -c1 etc. (default 0)
 */

#include <exec/types.h>

#include "iffp/iff.h"

/* local function prototypes */

LONG stdio_stream(struct Hook *, struct IFFHandle *, struct IFFStreamCmd *);

UBYTE *omodes[2] = {"r","w"};


/* openifile
 *
 * Passed a ParseInfo structure with a not-in-use IFFHandle, filename
 *   ("-c" or -cUnit like "-c1" for clipboard), and IFF open mode
 *   (IFFF_READ or IFFF_WRITE) opens file or clipboard for use with
 *   iffparse.library support modules.
 *
 * Returns 0 for success or an IFFERR (libraries/iffparse.h)
 */

LONG openifile(struct ParseInfo *pi, UBYTE *filename, ULONG iffopenmode)
{
        struct IFFHandle        *iff;
        BOOL    cboard;
        ULONG   unit = PRIMARY_CLIP;
        LONG    error;

        if(!pi)                 return(CLIENT_ERROR);
        if(!(iff=pi->iff))      return(CLIENT_ERROR);

        cboard = (*filename == '-'  &&  filename[1] == 'c');
        if(cboard && filename[2])       unit = atoi(&filename[2]);

        if (cboard)
                {
                /*
                 * Set up IFFHandle for Clipboard I/O.
                 */
                pi->clipboard = TRUE;
                if (!(iff->iff_Stream =
                                (ULONG)OpenClipboard(unit)))
                        {
                        message("Clipboard open of unit %ld failed.\n",unit);
                        return(NOFILE);
                        }
                InitIFFasClip(iff);
                }
        else
                {
                pi->clipboard = FALSE;
                /*
                 * Set up IFFHandle for buffered stdio I/O.
                 */
                if (!(iff->iff_Stream = (ULONG)
                   fopen(filename, omodes[iffopenmode & 1])))
                        {
                        message("%s: File open failed.\n",filename);
                        return(NOFILE);
```

```
                        }
                else initiffasstdio(iff);
                }

        D(bug("%s file opened: \n", cboard ? "[Clipboard]" : filename));

        pi->filename = filename;

        error=OpenIFF(iff, iffopenmode);

        pi->opened = error ? FALSE : TRUE;      /* currently open handle */

        D(bug("OpenIFF error = %ld\n",error));
        return(error);
}


/* closeifile
 *
 * closes file or clipboard opened with openifile, and frees all
 *   iffparse context parsed by parseifile.
 *
 * Note - You should closeifile as soon as possible if using clipboard
 *   ("-c[n]").  You also need to closeifile if, for example, you wish to
 *   reopen the file to write changes back out.  See the copychunks.c
 *   module for routines which allow you clone the chunks iffparse has
 *   gathered so that you can closeifile and still be able to modify and
 *   write back out gathered chunks.
 *
 */

void closeifile(struct ParseInfo *pi)
{
struct IFFHandle *iff;

        D(bug("closeifile:\n"));

        if(!pi)                 return;
        if(!(iff=pi->iff))      return;

        DD(bug("closeifile: About to CloseIFF if open, iff=$%lx, opened=%ld\n",
                        iff, pi->opened));

        if(pi->opened)  CloseIFF(iff);

        DD(bug("closeifile: About to close %s, stream=$%lx\n",
                        pi->clipboard ? "clipboard" : "file", iff->iff_Stream));
        if(iff->iff_Stream)
                {
                if (pi->clipboard)
                   CloseClipboard((struct ClipHandle *)(iff->iff_Stream));
                else
                   fclose ((FILE *)(iff->iff_Stream));
                }

        iff->iff_Stream = NULL;
        pi->clipboard = NULL;
        pi->opened = NULL;
}


/* parseifile
 *
 * Passed a ParseInfo with an initialized and open IFFHandle,
 *  grouptype (like ID_FORM), groupid (like ID_ILBM),
 *  and TAG_DONE terminated longword arrays of type,id
```

modules/parse.c Page 3

```
 *  for chunks to be grabbed, gathered, and stopped on
 *  (like { ID_ILBM, ID_BMHD, ID_ILBM, ID_CAMG, TAG_DONE })
 *  will parse an IFF file, grabbing/gathering and stopping
 *  on specified chunk.
 *
 * Note - you can call getcontext() (to continue after a stop chunk) or
 *  nextcontext() (after IFFERR_EOC, to parse next form in the same file)
 *  if you wish to continue parsing the same IFF file.  If parseifile()
 *  has to delve into a complex format to find your desired FORM, the
 *  pi->hunt flag will be set.  This should be a signal to you that
 *  you may not have the capability to simply modify and rewrite
 *  the data you have gathered.
 *
 * Returns 0 for success else and IFFERR (libraries/iffparse.h)
 */

LONG parseifile(pi,groupid,grouptype,propchks,collectchks,stopchks)
struct  ParseInfo *pi;
LONG groupid, grouptype;
LONG *propchks, *collectchks, *stopchks;
{
struct IFFHandle *iff;
register struct ContextNode     *cn;
LONG                    error = 0L;


        if(!(iff=pi->iff))      return(CLIENT_ERROR);

        if(!iff->iff_Stream)    return(IFFERR_READ);

        pi->hunt = FALSE;

        /*
         * Declare property, collection and stop chunks.
         */
        if (propchks)
          if (error = PropChunks (iff, propchks, chkcnt(propchks)))
                return (error);
        if (collectchks)
          if (error =
              CollectionChunks(iff, collectchks, chkcnt(collectchks)))
                return (error);
        if (stopchks)
          if (error = StopChunks (iff, stopchks, chkcnt(stopchks)))
                return (error);

        /*
         * We want to stop at the end of an ILBM context.
         */
        if (grouptype)
          if (error = StopOnExit (iff, grouptype, groupid))
                return(error);

        /*
         * Take first parse step to enter main chunk.
         */
        if (error = ParseIFF (iff, IFFPARSE_STEP))
                return(error);

        /*
         * Test the chunk info to see if simple form of type we want (ILBM).
         */
        if (!(cn = CurrentChunk (iff)))
                {
                /*
                 * This really should never happen.  If it does, it means
```

modules/parse.c Page 4

```
                 * our parser is broken.
                 */
                message("Parsing error; no top chunk!\n");
                return(NOFILE);
                }

        if (cn->cn_ID != groupid  ||  cn->cn_Type != grouptype)
                {

                D(bug("This is a(n) %.4s.%.4s.  Looking for embedded %.4s's...\n",
                  &cn->cn_Type, &cn->cn_ID, &grouptype));

                pi->hunt = TRUE;        /* Warning - this is a complex file */
                }

        if(!error)      error = getcontext(iff);
        return(error);
}

/* chkcnt
 *
 * simply counts the number of chunk pairs (type,id) in array
 */
LONG chkcnt(LONG *taggedarray)
{
LONG k = 0;

        while(taggedarray[k] != TAG_DONE) k++;
        return(k>>1);
}


/* currentchunkis
 *
 * returns the ID of the current chunk (like ID_CAMG)
 */
LONG currentchunkis(struct IFFHandle *iff, LONG type, LONG id)
{
register struct ContextNode     *cn;
LONG result = 0;

        if (cn = CurrentChunk (iff))
                {
                if((cn->cn_Type == type)&&(cn->cn_ID == id)) result = 1;
                }
        return(result);
}


/* contextis
 *
 * returns the enclosing context of the current chunk (like ID_ILBM)
 */
LONG contextis(struct IFFHandle *iff, LONG type, LONG id)
{
register struct ContextNode     *cn;
LONG result = 0;

       if (cn = (CurrentChunk (iff)))
           {
           if (cn = (ParentChunk(cn)))
               {
               if((cn->cn_Type == type)&&(cn->cn_ID == id)) result = 1;
               }
           }
        D(bug("This is a %.4s %.4s\n",&cn->cn_Type,&cn->cn_ID));

        return(result);
}


/* getcontext()
 *
 * Continues to gather the context which was specified to parseifile(),
 *  stopping at specified stop chunk, or end of context, or EOF
 *
 * Returns 0 (stopped on a stop chunk)
 *      or IFFERR_EOC (end of context, not an error)
 *      or IFFER_EOF (end of file)
 */
LONG getcontext(iff)
struct  IFFHandle *iff;
{
        LONG error = 0L;

        /* Based on our parse initialization,
         * ParseIFF() will return on a stop chunk (error = 0)
         * or end of context for an ILBM FORM (error = IFFERR_EOC)
         * or end of file (error = IFFERR_EOF)
         */
        return(error = ParseIFF(iff, IFFPARSE_SCAN));
}


/* nextcontext
 *
 * If you have finished parsing and reading your context (IFFERR_EOC),
 *   nextcontext will enter the next context contained in the file
 *   and parse it.
 *
 * Returns 0 or an IFFERR such as IFFERR_EOF (end of file)
 */

LONG nextcontext(iff)
struct  IFFHandle *iff;
{
        LONG error = 0L;

        error = ParseIFF(iff, IFFPARSE_STEP);

        D(bug("nextcontext: Got through next step\n"));

        return(error);
}


/* findpropdata
 *
 * finds specified chunk parsed from IFF file, and
 *   returns pointer to its sp_Data (or 0 for not found)
 */
UBYTE *findpropdata(iff, type, id)
struct IFFHandle        *iff;
LONG type, id;
        {
        register struct StoredProperty  *sp;

        if(sp = FindProp (iff, type, id)) return(sp->sp_Data);
        return(0);
        }
```

```
/*
 * File I/O hook functions which the IFF library will call.
 * A return of 0 indicates success (no error).
 *
 * Iffparse.library calls this code via struct Hook and Hook.asm
 */
static LONG
stdio_stream (hook, iff, actionpkt)
struct Hook             *hook;
struct IFFHandle        *iff;
struct IFFStreamCmd     *actionpkt;
{
        register FILE   *stream;
        register LONG   nbytes;
        register int    actual;
        register UBYTE  *buf;
        long    len;

        stream  = (FILE *) iff->iff_Stream;
        if(!stream)     return(1);

        nbytes  = actionpkt->sc_NBytes;
        buf     = (UBYTE *) actionpkt->sc_Buf;

        switch (actionpkt->sc_Command) {
        case IFFSCC_READ:
                do {
                        actual = nbytes > 32767 ? 32767 : nbytes;
                        if ((len=fread (buf, 1, actual, stream)) != actual)
                                break;
                        nbytes -= actual;
                        buf += actual;
                } while (nbytes > 0);
                return (nbytes ? IFFERR_READ : 0 );

        case IFFSCC_WRITE:
                do {
                        actual = nbytes > 32767 ? 32767 : nbytes;
                        if ((len=fwrite (buf, 1, actual, stream)) != actual)
                                break;
                        nbytes -= actual;
                        buf += actual;
                } while (nbytes > 0);
                return (nbytes ? IFFERR_WRITE : 0);

        case IFFSCC_SEEK:
                return ((fseek (stream, nbytes, 1) == -1) ? IFFERR_SEEK : 0);

        default:
                /*  No _INIT or _CLEANUP required.  */
                return (0);
        }
}

/* initiffasstdio (ie. init iff as stdio)
 *
 * sets up hook callback for the file stream handler above
 */
void initiffasstdio (iff)
struct IFFHandle *iff;
{
        extern LONG             HookEntry();
        static struct Hook      stdiohook = {
                { NULL },
                (ULONG (*)()) HookEntry,
                (ULONG (*)()) stdio_stream,
                NULL
        };

        /*
         * Initialize the IFF structure to point to the buffered I/O
         * routines.  Unbuffered I/O is terribly slow.
         */
        InitIFF (iff, IFFF_FSEEK | IFFF_RSEEK, &stdiohook);
}


/*
 * IFFerr
 *
 * Returns pointer to IFF Error string or NULL (no error)
 */
UBYTE *IFFerr(error)
LONG    error;
{
        /*
         * English error messages for possible IFFERR_#? returns from various
         * IFF routines.  To get the index into this array, take your IFFERR
         * code, negate it, and subtract one.
         *  idx = -error - 1;
         */
        static UBYTE    *errormsgs[] = {
                "End of file (not an error).",
                "End of context (not an error).",
                "No lexical scope.",
                "Insufficient memory.",
                "Stream read error.",
                "Stream write error.",
                "Stream seek error.",
                "File is corrupt.",
                "IFF syntax error.",
                "Not an IFF file.",
                "Required hook vector missing.",
                "Return to client."
        };
        static UBYTE unknown[32];
        static UBYTE client[] = "Client error";
        static UBYTE nofile[] = "File not found or wrong type";

        if (error < 0)
                {
                return(errormsgs[(-error) - 1]);
                }
        else if(error = CLIENT_ERROR)
                {
                return(client);
                }
        else if(error = NOFILE)
                {
                return(nofile);
                }
        else if(error)
                {
                sprintf(unknown,"Unknown error %ld",error);
                return(unknown);
                }
        else return(NULL);
}


/*
```

```
 * PutCk
 *
 * Writes one chunk of data to an iffhandle
 *
 */
long PutCk(struct IFFHandle *iff, long id, long size, void *data)
     {
     long error = 0, wlen;

     D(bug("PutCk: asked to push chunk \"%.4s\" ($%lx) length %ld\n",&id,id,size));

     if(error=PushChunk(iff, 0, id, size))
          {
          D(bug("PutCk: PushChunk of %.4s, error = %s, size = %ld\n",
                id, IFFerr(error), id));
          }
     else
          {
          D(bug("PutCk: PushChunk of %.4s, error = %ld\n",&id, error));

          /* Write the actual data */
          if((wlen = WriteChunkBytes(iff,data,size)) != size)
               {
               D(bug("WriteChunkBytes error: size = %ld, wrote %ld\n",size,wlen));
               error = IFFERR_WRITE;
               }
          else error = PopChunk(iff);
          D(bug("PutCk: After PopChunk - error = %ld\n",error));
          }
     return(error);
     }
```

```
/* saveilbm.c 05/91  C. Scheppner CBM
 *
 * High-level ILBM save routines
 */

#include "iffp/ilbm.h"
#include "iffp/ilbmapp.h"

extern struct Library *GfxBase;

/* screensave.c
 *
 * Given an ILBMInfo with a  currently available (not in use)
 *   ParseInfo->iff IFFHandle, a screen pointer, filename, and
 *   optional chunklist, will save screen as an ILBM
 * The struct Chunk *chunklist1 and 2 are for chunks you wish written
 * out other than BMHD, CMAP, and CAMG (they will be screened out
 * because they are computed and written separately).
 *
 * Note -  screensave passes NULL for transparent color and mask
 *
 * Returns 0 for success or an IFFERR (libraries/iffparse.h)
 */
LONG screensave(struct ILBMInfo *ilbm,
                        struct Screen *scr,
                        struct Chunk *chunklist1, struct Chunk *chunklist2,
                        UBYTE *filename)
{
extern struct Library *GfxBase;
UWORD *colortable, count;
ULONG modeid;
LONG error;
int k;

    if(GfxBase->lib_Version >= 36)
        modeid=GetVPModeID(&scr->ViewPort);
    else
        modeid = scr->ViewPort.Modes & OLDCAMGMASK;

    count = scr->ViewPort.ColorMap->Count;
    if(colortable = (UWORD *)AllocMem(count << 1, MEMF_CLEAR))
        {
        for(k=0; k<count; k++)  colortable[k]=GetRGB4(scr->ViewPort.ColorMap,k);

        error = saveilbm(ilbm, &scr->BitMap, modeid,
                scr->Width, scr->Height, scr->Width, scr->Height,
                colortable, count, 4,
                mskNone, 0,
                chunklist1, chunklist2, filename);
        FreeMem(colortable,count << 1);
        }
    else error = IFFERR_NOMEM;
    return(error);
}


/* saveilbm
 *
 * Given an ILBMInfo with a currently available (not-in-use)
 *   ParseInfo->iff IFFHandle, a BitMap ptr,
 *   modeid, widths/heights, colortable, ncolors, bitspergun,
 *   masking, transparent color, optional chunklists, and filename,
 *   will save the bitmap as an ILBM.
 *
 *  if bitspergun=4,  colortable is words, each with nibbles 0RGB
 *  if bitspergun=8,  colortable is byte guns of RGBRGB etc. (like a CMAP)
```

```
 *  if bitspergun=32, colortable is ULONG guns of RGBRGB etc.
 *     Only the high eight bits of each gun will be written to CMAP.
 *     Four bit guns n will be saved as nn
 *
 * The struct Chunk *chunklist is for chunks you wish written
 * other than BMHD, CMAP, and CAMG (they will be screened out)
 * because they are calculated and written separately
 *
 * Returns 0 for success, or an IFFERR
 */
LONG saveilbm(struct ILBMInfo *ilbm,
                struct BitMap *bitmap, ULONG modeid,
                WORD width, WORD height, WORD pagewidth, WORD pageheight,
                APTR colortable, UWORD ncolors, UWORD bitspergun,
                WORD masking, WORD transparentColor,
                struct Chunk *chunklist1, struct Chunk *chunklist2,
                UBYTE *filename)
{
struct IFFHandle *iff;
struct Chunk *chunk;
ULONG chunkID;
UBYTE *bodybuf;
LONG size, error = 0L;
#define BODYBUFSZ       4096

    iff = ilbm->ParseInfo.iff;

    if(!(modeid & 0xFFFF0000))  modeid &= OLDCAMGMASK;

    if(!(bodybuf = AllocMem(BODYBUFSZ,MEMF_PUBLIC)))
        {
        message("Not enough memory\n");
        return(IFFERR_NOMEM);
        }

    if(!(error = openifile(ilbm, filename, IFFF_WRITE)))
        {
        D(bug("Opened %s for write\n",filename));

        error = PushChunk(iff, ID_ILBM, ID_FORM, IFFSIZE_UNKNOWN);

        D(bug("After PushChunk FORM ILBM - error = %ld\n", error));

        initbmhd(&ilbm->Bmhd, bitmap, masking, cmpByteRun1, transparentColor,
                        width, height, pagewidth, pageheight, modeid);

        D(bug("Error before putbmhd = %ld\n",error));

        CkErr(putbmhd(iff,&ilbm->Bmhd));

        if(colortable)  CkErr(putcmap(iff,colortable,ncolors,bitspergun));

        ilbm->camg = modeid;
        D(bug("before putcamg - error = %ld\n",error));
        CkErr(putcamg(iff,&modeid));

        D(bug("Past putBMHD, CMAP, CAMG - error = %ld\n",error));

        /* Write out chunklists 1 & 2 (if any), except for
         * any BMHD, CMAP, or CAMG (computed/written separately)
         */
        for(chunk = chunklist1; chunk; chunk = chunk->ch_Next)
            {
            D(bug("chunklist1 - have a %.4s\n",&chunk->ch_ID));
            chunkID = chunk->ch_ID;
            if((chunkID != ID_BMHD)&&(chunkID != ID_CMAP)&&(chunkID != ID_CAMG))
```

```
                {
                size = chunk->ch_Size==IFFSIZE_UNKNOWN ?
                        strlen(chunk->ch_Data) : chunk->ch_Size;
                D(bug("Putting %.4s\n",&chunk->ch_ID));
                CkErr(PutCk(iff, chunkID, size, chunk->ch_Data));
                }
            }

        for(chunk = chunklist2; chunk; chunk = chunk->ch_Next)
            {
            chunkID = chunk->ch_ID;
            D(bug("chunklist2 - have a %.4s\n",&chunk->ch_ID));
            if((chunkID != ID_BMHD)&&(chunkID != ID_CMAP)&&(chunkID != ID_CAMG))
                {
                size = chunk->ch_Size==IFFSIZE_UNKNOWN ?
                        strlen(chunk->ch_Data) : chunk->ch_Size;
                D(bug("Putting %.4s\n",&chunk->ch_ID));
                CkErr(PutCk(iff, chunkID, size, chunk->ch_Data));
                }
            }

        /* Write out the BODY
         */
        CkErr(putbody(iff, bitmap, NULL, &ilbm->Bmhd, bodybuf, BODYBUFSZ));

        D(bug("Past putbody - error = %ld\n",error));


        CkErr(PopChunk(iff));    /* close out the FORM */
        closeifile(ilbm);        /* and the file */
        }

    FreeMem(bodybuf,BODYBUFSZ);

    return(error);
}
```

```
/* screen.c - 2.0 screen module for Display
 * based on scdemo, oscandemo, looki
 */

/*
Copyright (c) 1989, 1990 Commodore-Amiga, Inc.

Executables based on this information may be used in software
for Commodore Amiga computers. All other rights reserved.
This information is provided "as is"; no warranties are made.
All use is at your own risk, and no liability or responsibility
is assumed.
*/

#include "iffp/ilbmapp.h"

BOOL   VideoControlTags(struct ColorMap *,ULONG tags, ...);

extern struct Library *GfxBase;
extern struct Library *IntuitionBase;

struct TextAttr SafeFont = { (UBYTE *) "topaz.font", 8, 0, 0, };
UWORD  penarray[] = {~0};

/* default new window if none supplied in ilbm->nw */
struct   NewWindow      defnw = {
   0, 0,                                 /* LeftEdge and TopEdge */
   0, 0,                                 /* Width and Height */
   -1, -1,                               /* DetailPen and BlockPen */
   VANILLAKEY|MOUSEBUTTONS,              /* IDCMP Flags with Flags below */
   BACKDROP|BORDERLESS|SMART_REFRESH|NOCAREREFRESH|ACTIVATE|RMBTRAP,
   NULL, NULL,                           /* Gadget and Image pointers */
   NULL,                                 /* Title string */
   NULL,                                 /* Screen ptr null till opened */
   NULL,                                 /* BitMap pointer */
   50, 20,                               /* MinWidth and MinHeight */
   0 , 0,                                /* MaxWidth and MaxHeight */
   CUSTOMSCREEN                          /* Type of window */
   };


/* opendisplay - passed ILBMInfo, dimensions, modeID
 *
 *    Attempts to open correct 2.0 modeID screen and window,
 *    else an old 1.3 mode screen and window.
 *
 * Returns *window or NULL.
 */

struct Window *opendisplay(struct ILBMInfo *ilbm,
                           SHORT wide, SHORT high, SHORT deep,
                           ULONG mode)
    {
    struct NewWindow newwin, *nw;

    closedisplay(ilbm);
    if(ilbm->scr = openidscreen(ilbm, wide, high, deep, mode))
        {
        nw = &newwin;
        if(ilbm->windef) *nw = *(ilbm->windef);
        else *nw = *(&defnw);
        nw->Screen      = ilbm->scr;

        D(bug("sizes: scr= %ld x %ld  passed= %ld x %ld\n",
            ilbm->scr->Width,ilbm->scr->Height,wide,high));
        nw->Width       = wide;
        nw->Height      = high;
        if (!(ilbm->win = OpenWindow(nw)))
            {
            closedisplay(ilbm);
            D(bug("Failed to open window."));
            }
        else
            {
            if(ilbm->win->Flags & BACKDROP)
                {
                ShowTitle(ilbm->scr, FALSE);
                ilbm->TBState = FALSE;
                }
            }
        }

    if(ilbm->scr)       /* nulled out by closedisplay if OpenWindow failed */
        {
        ilbm->vp  = &ilbm->scr->ViewPort;
        ilbm->srp = &ilbm->scr->RastPort;
        ilbm->wrp = ilbm->win->RPort;
        }
    return(ilbm->win);
    }


void closedisplay(struct ILBMInfo *ilbm)
    {
    if(ilbm)
        {
        if (ilbm->win)  CloseWindow(ilbm->win), ilbm->win = NULL;
        if (ilbm->scr)  CloseScreen(ilbm->scr), ilbm->scr = NULL;
        ilbm->vp  = NULL;
        ilbm->srp = ilbm->wrp = NULL;
        }
    }



/* openidscreen - ILBMInfo, dimensions, modeID
 *
 *    Attempts to open correct 2.0 modeID screen with centered
 *    overscan based on user's prefs,
 *    else old 1.3 mode screen.
 *
 * If ilbm->stype includes CUSTOMBITMAP, ilbm->brbitmap will be
 *      used as the screen's bitmap.
 * If ilbm->stags is non-NULL, these tags will be added to the
 *      end of the taglist.
 *
 * Returns *screen or NULL.
 */

struct Screen *openidscreen(struct ILBMInfo *ilbm,
                            SHORT wide, SHORT high, SHORT deep,
                            ULONG mode)
    {
    struct NewScreen ns;                        /* for old style OpenScreen */
    struct Rectangle spos, dclip, txto, stdo, maxo, uclip;  /* display rectangles *
/
    struct Rectangle *uclipp;
    struct Screen    *scr = NULL;
    LONG     error, trynew;
    ULONG    bitmaptag, passedtags;
    BOOL     vctl;
```

```
    if (trynew = ((((struct Library *)GfxBase)->lib_Version >= 36)&&
         (((struct Library *)IntuitionBase)->lib_Version >= 36)))
        {
        /* if >= v36, see if mode is available */
        if(error = ModeNotAvailable(mode))
            {
            D(bug("Mode $%08lx not available, error=%ld:\n",mode,error));
            /* if not available, try fall back mode */
            mode = modefallback(mode,wide,high,deep);
            error = ModeNotAvailable(mode);

            D(bug("$%08lx ModeNotAvailable=%ld:\n",mode,error));
            }

        if(error) trynew = FALSE;
        else trynew=((QueryOverscan(mode,&txto,OSCAN_TEXT))&&
                         (QueryOverscan(mode,&stdo,OSCAN_STANDARD))&&
                             (QueryOverscan(mode,&maxo,OSCAN_MAX)));
        }

    D(bug("\nILBM: w=%ld, h=%ld, d=%ld, mode=0x%08lx\n",
                wide,high,deep,mode));
    D(bug("OPEN: %s.\n",
        trynew  ? "Is >= 2.0 and mode available, trying OpenScreenTags"
                : "Not 2.0, doing old OpenScreen"));

    if(trynew)
        {
        /* If user clip type specified and available, use it */
        if(ilbm->Video) ilbm->ucliptype = OSCAN_VIDEO;
        if((ilbm->ucliptype)&&(QueryOverscan(mode,&uclip,ilbm->ucliptype)))
                uclipp = &uclip;
        else uclipp = NULL;

        clipit(wide,high,&spos,&dclip,&txto,&stdo,&maxo,uclipp);

        D(bug("Using dclip  %ld,%ld  to  %ld,%ld... width=%ld height=%ld\n",
                        dclip.MinX,dclip.MinY,dclip.MaxX,dclip.MaxY,
                        dclip.MaxX-dclip.MinX+1,dclip.MaxY-dclip.MinY+1));
        D(bug("spos->minx = %ld, spos->miny = %ld\n",spos.MinX,spos.MinY));
        D(bug("DEBUG: About to attempt OpenScreenTags\n"));

        bitmaptag = ((ilbm->brbitmap)&&(ilbm->stype & CUSTOMBITMAP)) ?
                SA_BitMap : TAG_IGNORE;
        passedtags = ilbm->stags ? TAG_MORE : TAG_IGNORE;

        scr=(struct Screen *)OpenScreenTags((struct NewScreen *)NULL,
                SA_DisplayID,   mode,
                SA_Type,        ilbm->stype,
                SA_Behind,      TRUE,
                SA_Top,         spos.MinY,
                SA_Left,        spos.MinX,
                SA_Width,       wide,
                SA_Height,      high,
                SA_Depth,       deep,
                SA_DClip,       &dclip,
                SA_AutoScroll,  ilbm->Autoscroll ? TRUE : FALSE,
                SA_Title,       ilbm->stitle,
                SA_Font,        &SafeFont,
                SA_Pens,        penarray,
                SA_ErrorCode,   &error,
                bitmaptag,      ilbm->brbitmap,
                passedtags,     ilbm->stags,
                TAG_DONE
                );
            D(bug("DEBUG: OpenScreenTags scr at 0x%lx\n",scr));

            if(scr)
                {
                if(ilbm->Notransb)
                    {
                    vctl=VideoControlTags(scr->ViewPort.ColorMap,
                                VTAG_BORDERNOTRANS_SET, TRUE,
                                TAG_DONE);

        D(bug("VideoControl to set bordernotrans, error = %ld\n",vctl));

                    MakeScreen(scr);
                    RethinkDisplay();
                    }
                }
            else modeErrorMsg(mode,error);
        }

    if(!scr)
        {
        /* ns initialization for 1.3 old style OpenScreen only
         */
        ns.LeftEdge = ns.TopEdge = 0;
        ns.Width        =       wide;
        ns.Height       =       high;
        ns.Depth        =       deep;
        ns.ViewModes    =       modefallback(mode,wide,high,deep);
        ns.DetailPen    =       0;
        ns.BlockPen     =       1;
        ns.Gadgets      =       NULL;
        ns.CustomBitMap =       ((ilbm->brbitmap)&&(ilbm->stype & CUSTOMBITMAP))
                                        ? ilbm->brbitmap : NULL;
        ns.Font         =       &SafeFont;
        ns.DefaultTitle =       ilbm->stitle;
        ns.Type         =       ilbm->stype & 0x01FF;  /* allow only 1.3 types */

        scr=(struct Screen *)OpenScreen(&ns);

        D(bug("DEBUG: ns.ViewModes=0x%lx, vp->Modes=0x%lx\n",
                ns.ViewModes,scr->ViewPort.Modes));
        D(bug("DEBUG: non-extended scr at 0x%lx (0=failure)\n",scr));
        }
    return(scr);
    }


/*
 * modefallback - passed a mode id, attempts to provide a
 *                suitable old mode to use instead
 */

/* for old 1.3 screens */
#define MODE_ID_MASK (LACE|HIRES|HAM|EXTRA_HALFBRITE)

ULONG modefallback(ULONG mode, SHORT wide, SHORT high, SHORT deep)
{
ULONG newmode;

    /* For now, simply masks out everything but old mode bits.
     * This is just a cheap way to get some kind of display open
     *   and may be totally invalid for future modes.
     * Should search the display database for a suitable mode
     * based on the specific needs of your application.
     */
```

```
    newmode = mode & MODE_ID_MASK;

    D(bug("Try 0x%08lx instead of 0x%08lx\n",newmode,mode));
    return(newmode);
}


/*
 * clipit - passed width and height of a display, and the text, std, and
 *          max overscan rectangles for the mode, clipit fills in the
 *          spos (screen pos) and dclip rectangles to use in centering.
 *          Centered around smallest containing user-editable oscan pref,
 *          with dclip confined to legal maxoscan limits.
 *          Screens which center such that their top is below text
 *          oscan top, will be moved up.
 *          If a non-null uclip is passed, that clip is used instead.
 */
void clipit(SHORT wide, SHORT high,
            struct Rectangle *spos, struct Rectangle *dclip,
            struct Rectangle *txto, struct Rectangle *stdo,
            struct Rectangle *maxo, struct Rectangle *uclip)
{
struct  Rectangle *besto;
SHORT   minx, maxx, miny, maxy;
SHORT   txtw, txth, stdw, stdh, maxw, maxh, bestw, besth;

    /* get the txt, std and max widths and heights */
    txtw = txto->MaxX - txto->MinX + 1;
    txth = txto->MaxY - txto->MinY + 1;
    stdw = stdo->MaxX - stdo->MinX + 1;
    stdh = stdo->MaxY - stdo->MinY + 1;
    maxw = maxo->MaxX - maxo->MinX + 1;
    maxh = maxo->MaxY - maxo->MinY + 1;

    if((wide <= txtw)&&(high <= txth))
        {
        besto = txto;
        bestw = txtw;
        besth = txth;

        D(bug("Best clip is txto\n"));
        }
    else
        {
        besto = stdo;
        bestw = stdw;
        besth = stdh;

        D(bug("Best clip is stdo\n"));
        }

    D(bug("TXTO: mnx=%ld mny=%ld mxx=%ld mxy=%ld  stdw=%ld stdh=%ld\n",
            txto->MinX,txto->MinY,txto->MaxX,txto->MaxY,txtw,txth));
    D(bug("STDO: mnx=%ld mny=%ld mxx=%ld mxy=%ld  stdw=%ld stdh=%ld\n",
            stdo->MinX,stdo->MinY,stdo->MaxX,stdo->MaxY,stdw,stdh));
    D(bug("MAXO: mnx=%ld mny=%ld mxx=%ld mxy=%ld  maxw=%ld maxh=%ld\n",
            maxo->MinX,maxo->MinY,maxo->MaxX,maxo->MaxY,maxw,maxh));

    if(uclip)
        {
        *dclip = *uclip;
        spos->MinX = uclip->MinX;
        spos->MinY = uclip->MinY;

        D(bug("UCLIP: mnx=%ld mny=%ld maxx=%ld maxy=%ld\n",
                    dclip->MinX,dclip->MinY,dclip->MaxX,dclip->MaxY));
        }
    else
        {
        /* CENTER the screen based on best oscan prefs
        * but confine dclip within max oscan limits
        *
        * FIX MinX first */
        spos->MinX = minx = besto->MinX - ((wide - bestw) >> 1);
        maxx = wide + minx - 1;
        if(maxx > maxo->MaxX)  maxx = maxo->MaxX;       /* too right */
        if(minx < maxo->MinX)  minx = maxo->MinX;       /* too left  */

        D(bug("DCLIP: minx adjust from %ld to %ld\n",spos->MinX,minx));

        /* FIX MinY */
        spos->MinY = miny = besto->MinY - ((high - besth) >> 1);
        /* if lower than top of txto, move up */
        spos->MinY = miny = MIN(spos->MinY,txto->MinY);
        maxy = high + miny - 1;
        if(maxy > maxo->MaxY)  maxy = maxo->MaxY;       /* too down  */
        if(miny < maxo->MinY)  miny = maxo->MinY;       /* too up    */

        D(bug("DCLIP: miny adjust from %ld to %ld\n",spos->MinY,miny));

        /* SET up dclip */
        dclip->MinX = minx;
        dclip->MinY = miny;
        dclip->MaxX = maxx;
        dclip->MaxY = maxy;

        D(bug("CENTER: mnx=%ld mny=%ld maxx=%ld maxy=%ld\n",
                        dclip->MinX,dclip->MinY,dclip->MaxX,dclip->MaxY));
        }
}


void modeErrorMsg(ULONG mode, ULONG errorcode)
   {
   UBYTE *s=NULL;

        D(bug("DEBUG: Can't open mode ID 0x%08lx screen: ",mode));

        switch ( errorcode )
        {
        case OSERR_NOMEM:
            s="Not enough memory.";
            break;
        case OSERR_NOCHIPMEM:
            s="Not enough chip memory.";
            break;
#ifdef DEBUG
        case OSERR_NOMONITOR:
            s="monitor not available.";
            break;

        case OSERR_NOCHIPS:
            s="new chipset not available.";
            break;

        case OSERR_PUBNOTUNIQUE:
            s="public screen already open.";
            break;
        case OSERR_UNKNOWNMODE:
            s="mode ID is unknown.";
            break;
        default:
```

```
            message("unknown mode error %ld\n",errorcode);
#endif
        }
    if(s) message("%s\n",s);
    }


/*----------------------------------------------------------------------*/

BOOL VideoControlTags(struct ColorMap *cm, ULONG tags, ...)
    {
    return (VideoControl(cm, (struct TagItem *)&tags));
    }


/*----------------------------------------------------------------------*/
```

modules/screendump.c   Page 1

```
/*
 * screendump.c    - routine to dump rastport (iffparse not required)
 *
 */

#include <exec/types.h>
#include <intuition/screens.h>
#include <devices/printer.h>

#ifndef NO_PROTOS
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#endif


/* screendump
 *
 * Passed a screen pointer, source x, source y, width, height,
 *   destcols and io_Special flags, will print part or all of a screen.
 *
 * If 0 is passed for BOTH destcols and special, screendump()
 *   assumes you want IT to compute suitable values.
 * In this case:
 *   1. If srcx and srcy are 0, and srcw and srch are same as
 *      screen width and height, screendump will set destcols=0,
 *      and special = SPECIAL_FULLCOLS|SPECIAL_ASPECT
 *      for a full width aspected dump.
 *
 *   2. If srcx or srcy are nonzero, or srcw or srch are different
 *      from screen width or height, screendump will print a
 *      fractional size dump relative to the size whole screendump
 *      would have been.
 *
 * Returns 0 for success or printer io_Error (devices/printer.h)
 */

int screendump(struct Screen *scr,
                UWORD srcx, UWORD srcy, UWORD srcw, UWORD srch,
                LONG destcols, UWORD iospecial)
    {
    struct IODRPReq *iodrp;
    struct MsgPort  *printerPort;
    struct ViewPort *vp;
    ULONG tmpl;
    int error = PDERR_BADDIMENSION;

    if(!scr)    return(error);

    if((!destcols)&&(!iospecial))
        {
        /* Then we compute what they should be */
        if((!srcx)&&(!srcy)&&(srcw==scr->Width)&&(srch==scr->Height))
            {
            iospecial = SPECIAL_FULLCOLS|SPECIAL_ASPECT;
            }
        else
            {
            iospecial = SPECIAL_FRACCOLS|SPECIAL_ASPECT;
            tmpl = srcw;
            tmpl = tmpl << 16;
            destcols = (tmpl / scr->Width) << 16;
            }
        }

    if(printerPort = CreatePort(0,0))
        {
```

modules/screendump.c   Page 2

```
        if(iodrp=
           (struct IODRPReq *)CreateExtIO(printerPort,sizeof(struct IODRPReq)))
            {
            if(!(error=OpenDevice("printer.device",0,iodrp,0)))
                {
                vp = &scr->ViewPort;
                iodrp->io_Command = PRD_DUMPRPORT;
                iodrp->io_RastPort = &scr->RastPort;
                iodrp->io_ColorMap = vp->ColorMap;
                iodrp->io_Modes = (ULONG)vp->Modes;
                iodrp->io_SrcX = srcx;
                iodrp->io_SrcY = srcy;
                iodrp->io_SrcWidth = srcw;
                iodrp->io_SrcHeight = srch;
                iodrp->io_DestCols = destcols;
        /*      iodrp->io_DestRows = 0; cleared by allocation */
                iodrp->io_Special = iospecial;

                error = DoIO(iodrp);

                CloseDevice(iodrp);
                }
            DeleteExtIO(iodrp);
            }
        DeletePort(printerPort);
        }
    return(error);
    }
```

modules/unpacker.c

```
#include "iffp/ilbm.h"
#include "iffp/packer.h"

/*----------------------------------------------------------------------*
 * unpacker.c Convert data from "cmpByteRun1" run compression. 11/15/85
 *
 * Based on code by Jerry Morrison and Steve Shaw, Electronic Arts.
 * This software is in the public domain.
 *
 *      control bytes:
 *       [0..127]   : followed by n+1 bytes of data.
 *       [-1..-127] : followed by byte to be repeated (-n)+1 times.
 *       -128       : NOOP.
 *
 * This version for the Commodore-Amiga computer.
 *----------------------------------------------------------------------*/

/*----------- UnPackRow ----------------------------------------------*/

#define UGetByte()      (*source++)
#define UPutByte(c)     (*dest++ = (c))

/* Given POINTERS to POINTER variables, unpacks one row, updating the source
 * and destination pointers until it produces dstBytes bytes.
 */

BOOL unpackrow(BYTE **pSource, BYTE **pDest, WORD srcBytes0, WORD dstBytes0)
    {
    register BYTE *source = *pSource;
    register BYTE *dest   = *pDest;
    register WORD n;
    register BYTE c;
    register WORD srcBytes = srcBytes0, dstBytes = dstBytes0;
    BOOL error = TRUE;  /* assume error until we make it through the loop */
    WORD minus128 = -128;  /* get the compiler to generate a CMP.W */

    while( dstBytes > 0 )  {
        if ( (srcBytes -= 1) < 0 )  goto ErrorExit;
        n = UGetByte();

        if (n >= 0) {
            n += 1;
            if ( (srcBytes -= n) < 0 )  goto ErrorExit;
            if ( (dstBytes -= n) < 0 )  goto ErrorExit;
            do {  UPutByte(UGetByte());  } while (--n > 0);
            }

        else if (n != minus128) {
            n = -n + 1;
            if ( (srcBytes -= 1) < 0 )  goto ErrorExit;
            if ( (dstBytes -= n) < 0 )  goto ErrorExit;
            c = UGetByte();
            do {  UPutByte(c);  } while (--n > 0);
            }
        }
    error = FALSE;      /* success! */

  ErrorExit:
    *pSource = source;  *pDest = dest;
    return(error);
    }

/* end */
```

other/clipftxt.c

```
;/* clipftxt.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -j73 clipftxt.c
Blink FROM LIB:c.o,clipftxt.o TO clipftxt LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

*
* clipftxt.c:   Writes ASCII text to clipboard unit as FTXT
*               (All clipboard data must be IFF)
*
* Usage: clipftxt unitnumber
*
* To convert to an example of reading only, comment out #define WRITEREAD
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/iffparse.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/iffparse_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

/* Causes example to write FTXT first, then read it back
 * Comment out to create a reader only
 */
#define WRITEREAD


#define MINARGS 2

/* 2.0 Version string for c:Version to find */
UBYTE vers[] = "\0$VER: clipftxt 37.2";

UBYTE usage[] = "Usage: clipftxt unitnumber (use zero for primary unit)";

/*
 * Text error messages for possible IFFERR_#? returns from various
 * IFF routines.  To get the index into this array, take your IFFERR code,
 * negate it, and subtract one.
 *  idx = -error - 1;
 */
char    *errormsgs[] = {
        "End of file (not an error).",
        "End of context (not an error).",
        "No lexical scope.",
        "Insufficient memory.",
        "Stream read error.",
        "Stream write error.",
        "Stream seek error.",
        "File is corrupt.",
        "IFF syntax error.",
        "Not an IFF file.",
        "Required call-back hook missing.",
        "Return to client.  You should never see this."
};

#define RBUFSZ 512
```

other/clipftxt.c Page 2

```
#define  ID_FTXT        MAKE_ID('F','T','X','T')
#define  ID_CHRS        MAKE_ID('C','H','R','S')

struct Library *IFFParseBase;

UBYTE mytext[]="This FTXT written to clipboard by clipftxt example.\n";

void main(int argc, char **argv)
{
    struct IFFHandle    *iff = NULL;
    struct ContextNode  *cn;
    long                error=0, unitnumber=0, rlen;
    int textlen;
    UBYTE readbuf[RBUFSZ];

        /* if not enough args or '?', print usage */
        if(((argc)&&(argc<MINARGS))||(argv[argc-1][0]=='?'))
                {
                printf("%s\n",usage);
                exit(RETURN_WARN);
                }

        unitnumber = atoi(argv[1]);

        if (!(IFFParseBase = OpenLibrary ("iffparse.library", 0L)))
                {
                puts("Can't open iff parsing library.");
                goto bye;
                }

        /*
         * Allocate IFF_File structure.
         */
        if (!(iff = AllocIFF ()))
                {
                puts ("AllocIFF() failed.");
                goto bye;
                }

        /*
         * Set up IFF_File for Clipboard I/O.
         */
        if (!(iff->iff_Stream = (ULONG) OpenClipboard (unitnumber)))
                {
                puts ("Clipboard open failed.");
                goto bye;
                }
        else printf("Opened clipboard unit %ld\n",unitnumber);

        InitIFFasClip (iff);

#ifdef WRITEREAD

        /*
         * Start the IFF transaction.
         */
        if (error = OpenIFF (iff, IFFF_WRITE))
                {
                puts ("OpenIFF for write failed.");
                goto bye;
                }

        /*
         * Write our text to the clipboard as CHRS chunk in FORM FTXT
         *
```

other/clipftxt.c Page 3

```
         * First, write the FORM ID (FTXT)
         */
        if(!(error=PushChunk(iff, ID_FTXT, ID_FORM, IFFSIZE_UNKNOWN)))
                {
                /* Now the CHRS chunk ID followed by the chunk data
                 * We'll just write one CHRS chunk.
                 * You could write more chunks.
                 */
                if(!(error=PushChunk(iff, 0, ID_CHRS, IFFSIZE_UNKNOWN)))
                        {
                        /* Now the actual data (the text) */
                        textlen = strlen(mytext);
                        if(WriteChunkBytes(iff, mytext, textlen) != textlen)
                                {
                                puts("Error writing CHRS data.");
                                error = IFFERR_WRITE;
                                }
                        }
                if(!error) error = PopChunk(iff);
                }
        if(!error) error = PopChunk(iff);


        if(error)
                {
                printf ("IFF write failed, error %ld: %s\n",
                        error, errormsgs[-error - 1]);
                goto bye;
                }
        else printf("Wrote text to clipboard as FTXT\n");

        /*
         * Now let's close it, then read it back
         * First close the write handle, then close the clipboard
         */
        CloseIFF(iff);
        if (iff->iff_Stream) CloseClipboard ((struct ClipboardHandle *)
                                             iff->iff_Stream);

        if (!(iff->iff_Stream = (ULONG) OpenClipboard (unitnumber)))
                {
                puts ("Reopen of Clipboard failed.");
                goto bye;
                }
        else printf("Reopened clipboard unit %ld\n",unitnumber);

#endif /* WRITEREAD */

        if (error = OpenIFF (iff, IFFF_READ))
                {
                puts ("OpenIFF for read failed.");
                goto bye;
                }

        /* Tell iffparse we want to stop on FTXT CHRS chunks */
        if (error = StopChunk(iff, ID_FTXT, ID_CHRS))
                {
                puts ("StopChunk failed.");
                goto bye;
                }

        /* Find all of the FTXT CHRS chunks */
        while(1)
                {
                error = ParseIFF(iff,IFFPARSE_SCAN);
                if(error == IFFERR_EOC) continue;       /* enter next context */
                else if(error) break;

                /* We only asked to stop at FTXT CHRS chunks
                 * If no error we've hit a stop chunk
                 * Read the CHRS chunk data
                 */
                cn = CurrentChunk(iff);

                if((cn)&&(cn->cn_Type == ID_FTXT)&&(cn->cn_ID == ID_CHRS))
                        {
                        printf("CHRS chunk contains:\n");
                        while((rlen = ReadChunkBytes(iff,readbuf,RBUFSZ)) > 0)
                                {
                                Write(Output(),readbuf,rlen);
                                }
                        if(rlen < 0)    error = rlen;
                        }
                }

        if((error)&&(error != IFFERR_EOF))
                {
                printf ("IFF read failed, error %ld: %s\n",
                        error, errormsgs[-error - 1]);
                }

bye:
        if (iff) {
                /*
                 * Terminate the IFF transaction with the stream.  Free
                 * all associated structures.
                 */
                CloseIFF (iff);

                /*
                 * Close the clipboard stream
                 */
                if (iff->iff_Stream)
                                CloseClipboard ((struct ClipboardHandle *)
                                                iff->iff_Stream);
                /*
                 * Free the IFF_File structure itself.
                 */
                FreeIFF (iff);
                }
        if (IFFParseBase)       CloseLibrary (IFFParseBase);

        exit (RETURN_OK);
}
```

```
/*
 * cycvb.c --- Dan Silva's DPaint color cycling interrupt code
 *
 *    Use this fragment as an example for interrupt driven color cycling
 *    If compiled with SAS, include flags -v -y on LC2
 */

#include <exec/types.h>
#include <exec/interrupts.h>
#include <graphics/view.h>
#include <iff/compiler.h>

#define MAXNCYCS 4
#define NO  FALSE
#define YES TRUE
#define LOCAL static

typedef struct {
    SHORT count;
    SHORT rate;
    SHORT flags;
    UBYTE low, high;  /* bounds of range */
    } Range;

/* Range flags values */
#define RNG_ACTIVE   1
#define RNG_REVERSE 2
#define RNG_NORATE 36  /* if rate == NORATE, don't cycle */

/* cycling frame rates */
#define OnePerTick   16384
#define OnePerSec    OnePerTick/60

extern Range  cycles[];
extern BOOL   cycling[];
extern WORD   cycols[];
extern struct ViewPort *vport;
extern SHORT  nColors;


MyVBlank()  {
   int i,j;
   LOCAL  Range *cyc;
   LOCAL  WORD temp;
   LOCAL  BOOL anyChange;

#ifdef IS_AZTEC
#asm
       movem.l  a2-a7/d2-d7,-(sp)
       move.l   a1,a4
#endasm
#endif

   if (cycling)  {
      anyChange = NO;
      for (i=0; i<MAXNCYCS; i++)  {
         cyc = &cycles[i];
         if ( (cyc->low == cyc->high) ||
              ((cyc->flags&RNG_ACTIVE) == 0) ||
              (cyc->rate == RNG_NORATE) )
                continue;

         cyc->count += cyc->rate;
         if (cyc->count >= OnePerTick)  {
            anyChange = YES;
            cyc->count -= OnePerTick;
```

other/cycvb.c

```
            if (cyc->flags&RNG_REVERSE)  {
               temp = cycols[cyc->low];
               for (j=cyc->low; j < cyc->high; j++)
                  cycols[j] = cycols[j+1];
               cycols[cyc->low] = temp;
               }
            else  {
               temp = cycols[cyc->high];
               for (j=cyc->high; j > cyc->low; j--)
                  cycols[j] = cycols[j-1];
               cycols[cyc->low] = temp;
               }
            }
         }
      if (anyChange) LoadRGB4(vport,cycols,nColors);
      }

#ifdef IS_AZTEC
      ;  /* this is necessary */
#asm
      movem.l   (sp)+,a2-a7/d2-d7
#endasm
#endif

   return(0);  /* interrupt routines have to do this */
   }


/*
 *  Code to install/remove cycling interrupt handler
 */

LOCAL char myname[] = "MyVB";  /* Name of interrupt handler */
LOCAL struct Interrupt intServ;

typedef void (*VoidFunc)();

StartVBlank()  {
#ifdef IS_AZTEC
   intServ.is_Data = GETAZTEC();  /* returns contents of register a4 */
#else
   intServ.is_Data = NULL;
#endif
   intServ.is_Code = (VoidFunc)&MyVBlank;
   intServ.is_Node.ln_Succ = NULL;
   intServ.is_Node.ln_Pred = NULL;
   intServ.is_Node.ln_Type = NT_INTERRUPT;
   intServ.is_Node.ln_Pri  = 0;
   intServ.is_Node.ln_Name = myname;
   AddIntServer(5,&intServ);
   }

StopVBlank() { RemIntServer(5,&intServ); }

/**/
```

other/ilbmscan.c

```
;/* ilbmscan.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -j73 ilbmscan.c
Blink FROM LIB:c.o,ilbmscan.o TO ilbmscan LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

*
* ilbmscan.c:   Prints the size, aspect, mode, etc. of ILBM's
*               Scans through an IFF file for all ILBM's
*
* Usage: ilbmscan -c              ; For clipboard scanning
*    or  ilbmscan <file>          ; For DOS file scanning
*
* Based on sift.c by by Stuart Ferguson and Leo Schwab
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/iffparse.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/iffparse_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

/*  The structure of a FORM ILBM 'BMHD' and 'CAMG' chunks
 *  Such structures are defined in the spec for a FORM
 *  and may also be provided in include files
 */
/*  Bitmap header (BMHD) structure  */
typedef struct {
        UWORD   w, h;            /*  Width, height in pixels */
        WORD    x, y;            /*  x, y position for this bitmap  */
        UBYTE   nplanes;         /*  # of planes  */
        UBYTE   Masking;
        UBYTE   Compression;
        UBYTE   pad1;
        UWORD   TransparentColor;
        UBYTE   XAspect, YAspect;
        WORD    PageWidth, PageHeight;
} BitMapHeader;

/* Commodore Amiga (CAMG) Viewmodes structure */
typedef struct {
   ULONG ViewModes;
   } CamgChunk;

#define ID_ILBM         MAKE_ID('I','L','B','M')
#define ID_BMHD         MAKE_ID('B','M','H','D')
#define ID_CMAP         MAKE_ID('C','M','A','P')
#define ID_CAMG         MAKE_ID('C','A','M','G')
#define ID_BODY         MAKE_ID('B','O','D','Y')

void PrintILBMInfo(struct IFFHandle *);

#define MINARGS 2

/* 2.0 Version string for c:Version to find */
UBYTE vers[] = "\0$VER: ilbmscan 37.3";
UBYTE usage[] = "Usage: ilbmscan IFFfilename (or -c for clipboard)";

/*
 * Text error messages for possible IFFERR_#? returns from various
 * IFF routines.  To get the index into this array, take your IFFERR code,
 * negate it, and subtract one.
 *  idx = -error - 1;
 */
char    *errormsgs[] = {
        "End of file (not an error).",
        "End of context (not an error).",
        "No lexical scope.",
        "Insufficient memory.",
        "Stream read error.",
        "Stream write error.",
        "Stream seek error.",
        "File is corrupt.",
        "IFF syntax error.",
        "Not an IFF file.",
        "Required call-back hook missing.",
        "Return to client.  You should never see this."
};

struct Library *IFFParseBase;


void main(int argc, char **argv)
{
    struct IFFHandle    *iff = NULL;
    long                error;
    short               cbio;

        /* if not enough args or '?', print usage */
        if(((argc)&&(argc<MINARGS))||(argv[argc-1][0]=='?'))
                {
                printf("%s\n",usage);
                exit(RETURN_OK);
                }

        /*
         * Check to see if we are doing I/O to the Clipboard.
         */
        cbio = (argv[1][0] == '-'  &&  argv[1][1] == 'c');

        if (!(IFFParseBase = OpenLibrary ("iffparse.library", 0L)))
                {
                printf("Can't open iff parsing library.");
                goto bye;
                }

        /*
         * Allocate IFF_File structure.
         */
        if (!(iff = AllocIFF ()))
                {
                printf ("AllocIFF() failed.");
                goto bye;
                }

        /*
         * Internal support is provided for both AmigaDOS files, and the
         * clipboard.device.  This bizarre 'if' statement performs the
         * appropriate machinations for each case.
         */
        if (cbio)
```

```
                {
                /*
                 * Set up IFF_File for Clipboard I/O.
                 */
                if (!(iff->iff_Stream =
                                (ULONG) OpenClipboard (PRIMARY_CLIP)))
                        {
                        printf("Clipboard open failed.");
                        goto bye;
                        }
                InitIFFasClip (iff);
                }
        else
                {
                /*
                 * Set up IFF_File for AmigaDOS I/O.
                 */
                if (!(iff->iff_Stream = Open (argv[1], MODE_OLDFILE)))
                        {
                        printf("File open failed.");
                        goto bye;
                        }
                InitIFFasDOS (iff);
                }

        /*
         * Start the IFF transaction.
         */
        if (error = OpenIFF (iff, IFFF_READ))
                {
                printf("OpenIFF failed.");
                goto bye;
                }

        /* We want to collect BMHD and CAMG */
        PropChunk(iff, ID_ILBM, ID_BMHD);
        PropChunk(iff, ID_ILBM, ID_CAMG);
        PropChunk(iff, ID_ILBM, ID_CMAP);

        /* Stop at the BODY */
        StopChunk(iff, ID_ILBM, ID_BODY);

        /* And let us know (IFFERR_EOC) when leaving a FORM ILBM */
        StopOnExit(iff,ID_ILBM, ID_FORM);

        /* Do the scan.
         * The while(1) will let us delve into more complex formats
         * to find FORM ILBM's
         */
        while (1)
                {
                error = ParseIFF(iff, IFFPARSE_SCAN);
                /*
                 * Since we're only interested in when we enter a context,
                 * we "discard" end-of-context (_EOC) events.
                 */
                if (error == IFFERR_EOC)
                        {
                        printf("Exiting FORM ILBM\n\n");
                        continue;
                        }
                else if (error)
                        /*
                         * Leave the loop if there is any other error.
                         */
                        break;
                /*
                 * If we get here, error was zero
                 * Since we did IFFPARSE_SCAN, zero error should mean
                 * we are at our Stop Chunk (BODY)
                 */
                PrintILBMInfo(iff);
                }

        /*
         * If error was IFFERR_EOF, then the parser encountered the end of
         * the file without problems.  Otherwise, we print a diagnostic.
         */
        if (error == IFFERR_EOF)
                printf("File scan complete.\n");
        else
                printf("File scan aborted, error %ld: %s\n",
                        error, errormsgs[-error - 1]);

bye:
        if (iff) {
                /*
                 * Terminate the IFF transaction with the stream.  Free
                 * all associated structures.
                 */
                CloseIFF (iff);

                /*
                 * Close the stream itself.
                 */
                if (iff->iff_Stream)
                        if (cbio)
                                CloseClipboard ((struct ClipboardHandle *)
                                                iff->iff_Stream);
                        else
                                Close (iff->iff_Stream);

                /*
                 * Free the IFF_File structure itself.
                 */
                FreeIFF (iff);
                }
        if (IFFParseBase)       CloseLibrary (IFFParseBase);

        exit (RETURN_OK);
}


void
PrintILBMInfo(iff)
struct IFFHandle *iff;
{
        struct StoredProperty   *sp;
        BitMapHeader    *bmhd;
        CamgChunk       *camg;

        /*
         * Get a pointer to the stored propery BMHD
         */
        if (!(sp = FindProp(iff, ID_ILBM, ID_BMHD)))
                printf("No BMHD found\n");
        else
                {
                /* If property is BMHD, sp->sp_Data is ptr to data in BMHD */
                bmhd = (BitMapHeader *)sp->sp_Data;
                printf("BMHD: Width      = %ld\n",bmhd->w);
```

```
                printf("      Height     = %ld\n",bmhd->h);
                printf("      PageWidth  = %ld\n",bmhd->PageWidth);
                printf("      PageHeight = %ld\n",bmhd->PageHeight);
                printf("      nplanes    = %ld\n",bmhd->nplanes);
                printf("      Masking    = %ld\n",bmhd->Masking);
                printf("      Compression= %ld\n",bmhd->Compression);
                printf("      TransColor = %ld\n",bmhd->TransparentColor);
                printf("      X/Y Aspect = %ld/%ld\n",bmhd->XAspect,bmhd->YAspect);
                }

        /*
         * Get a pointer to the stored propery CMAP
         */
        if (!(sp = FindProp(iff, ID_ILBM, ID_CMAP)))
                printf("No CMAP found\n");
        else
                {
                /* If property is CMAP, sp->sp_Data is ptr to data in CMAP */
                printf("CMAP: contains RGB values for %ld registers\n",
                        sp->sp_Size / 3);
                }

        /*
         * Get a pointer to the stored propery CAMG
         */
        if (!(sp = FindProp(iff, ID_ILBM, ID_CAMG)))
                printf("No CAMG found\n");
        else
                {
                /* If property is CAMG, sp->sp_Data is ptr to data in CAMG */
                camg = (CamgChunk *)sp->sp_Data;
                printf("CAMG: ModeID     = $%08lx\n\n",camg->ViewModes);
                }
}
```

```
;/* sift.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -j73 sift.c
Blink FROM LIB:c.o,sift.o TO sift LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

*
* sift.c:       Takes any IFF file and tells you what's in it.  Verifies
*               syntax and all that cool stuff.
*
* Usage: sift -c                ; For clipboard scanning
*    or  sift <file>            ; For DOS file scanning
*
* Reads the specified stream and prints an IFFCheck-like listing of the
* contents of the IFF file, if any.  Stream is a DOS file for <file>
* argument, or is the clipboard's primary clip for -c.
* This program must be run from a CLI.
*
* Based on original sift.c by by Stuart Ferguson and Leo Schwab
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/iffparse.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/iffparse_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

#define MINARGS 2

/* 2.0 Version string for c:Version to find */
UBYTE vers[] = "\0$VER: sift 37.1";

UBYTE usage[] = "Usage: sift IFFfilename (or -c for clipboard)";

/* proto for our function */
void PrintTopChunk (struct IFFHandle *);

/*
 * Text error messages for possible IFFERR_#? returns from various
 * IFF routines.  To get the index into this array, take your IFFERR code,
 * negate it, and subtract one.
 *  idx = -error - 1;
 */
char    *errormsgs[] = {
        "End of file (not an error).",
        "End of context (not an error).",
        "No lexical scope.",
        "Insufficient memory.",
        "Stream read error.",
        "Stream write error.",
        "Stream seek error.",
        "File is corrupt.",
        "IFF syntax error.",
        "Not an IFF file.",
        "Required call-back hook missing.",
        "Return to client.  You should never see this."
```

```
};

struct Library *IFFParseBase;


void main(int argc, char **argv)
{
    struct IFFHandle    *iff = NULL;
    long                error;
    short               cbio;

        /* if not enough args or '?', print usage */
        if(((argc)&&(argc<MINARGS))||(argv[argc-1][0]=='?'))
                {
                printf("%s\n",usage);
                goto bye;
                }

        /*
         * Check to see if we are doing I/O to the Clipboard.
         */
        cbio = (argv[1][0] == '-'  &&  argv[1][1] == 'c');

        if (!(IFFParseBase = OpenLibrary ("iffparse.library", 0L)))
                {
                puts("Can't open iff parsing library.");
                goto bye;
                }

        /*
         * Allocate IFF_File structure.
         */
        if (!(iff = AllocIFF ()))
                {
                puts ("AllocIFF() failed.");
                goto bye;
                }

        /*
         * Internal support is provided for both AmigaDOS files, and the
         * clipboard.device.  This bizarre 'if' statement performs the
         * appropriate machinations for each case.
         */
        if (cbio)
                {
                /*
                 * Set up IFF_File for Clipboard I/O.
                 */
                if (!(iff->iff_Stream =
                                (ULONG) OpenClipboard (PRIMARY_CLIP)))
                        {
                        puts ("Clipboard open failed.");
                        goto bye;
                        }
                InitIFFasClip (iff);
                }
        else
                {
                /*
                 * Set up IFF_File for AmigaDOS I/O.
                 */
                if (!(iff->iff_Stream = Open (argv[1], MODE_OLDFILE)))
                        {
                        puts ("File open failed.");
                        goto bye;
                        }
                InitIFFasDOS (iff);
                }

        /*
         * Start the IFF transaction.
         */
        if (error = OpenIFF (iff, IFFF_READ))
                {
                puts ("OpenIFF failed.");
                goto bye;
                }

        while (1)
                {
                /*
                 * The interesting bit.  IFFPARSE_RAWSTEP permits us to
                 * have precision monitoring of the parsing process, which
                 * is necessary if we wish to print the structure of an
                 * IFF file.  ParseIFF() with _RAWSTEP will return the
                 * following things for the following reasons:
                 *
                 * Return code:                 Reason:
                 * 0                            Entered new context.
                 * IFFERR_EOC                   About to leave a context.
                 * IFFERR_EOF                   Encountered end-of-file.
                 * <anything else>              A parsing error.
                 */
                error = ParseIFF (iff, IFFPARSE_RAWSTEP);

                /*
                 * Since we're only interested in when we enter a context,
                 * we "discard" end-of-context (_EOC) events.
                 */
                if (error == IFFERR_EOC)
                        continue;
                else if (error)
                        /*
                         * Leave the loop if there is any other error.
                         */
                        break;

                /*
                 * If we get here, error was zero.
                 * Print out the current state of affairs.
                 */
                PrintTopChunk (iff);
                }

        /*
         * If error was IFFERR_EOF, then the parser encountered the end of
         * the file without problems.  Otherwise, we print a diagnostic.
         */
        if (error == IFFERR_EOF)
                puts ("File scan complete.");
        else
                printf ("File scan aborted, error %ld: %s\n",
                        error, errormsgs[-error - 1]);

bye:
        if (iff) {
                /*
                 * Terminate the IFF transaction with the stream.  Free
                 * all associated structures.
                 */
                CloseIFF (iff);
```

```
                /*
                 * Close the stream itself.
                 */
                if (iff->iff_Stream)
                        if (cbio)
                                CloseClipboard ((struct ClipboardHandle *)
                                                iff->iff_Stream);
                        else
                                Close (iff->iff_Stream);

                /*
                 * Free the IFF_File structure itself.
                 */
                FreeIFF (iff);
                }
        if (IFFParseBase)       CloseLibrary (IFFParseBase);

        exit (RETURN_OK);
}


void
PrintTopChunk (iff)
struct IFFHandle *iff;
{
        struct ContextNode      *top;
        short                   i;
        char                    idbuf[5];

        /*
         * Get a pointer to the context node describing the current context.
         */
        if (!(top = CurrentChunk (iff)))
                return;

        /*
         * Print a series of dots equivalent to the current nesting depth of
         * chunks processed so far.  This will cause nested chunks to be
         * printed out indented.
         */
        for (i = iff->iff_Depth;  i--; )
                printf (". ");

        /*
         * Print out the current chunk's ID and size.
         */
        printf ("%s %ld ", IDtoStr (top->cn_ID, idbuf), top->cn_Size);

        /*
         * Print the current chunk's type, with a newline.
         */
        puts (IDtoStr (top->cn_Type, idbuf));
}
```

appendix B
EXAMPLE DEVICE

This appendix contains source code for a sample device. The example code is an excellent starting point for those who want to create a custom device and add it to the Amiga's system software.

The example is a complete four-unit, static-sized RAM disk that works under the old (standard) filing system, the new Fast Filing System (FFS), and has optional code to bind it to an AUTOCONFIG^{TM} device.

The examples have been assembled under the Metacomco assembler V11.0 and under the CAPE assembler V2.0.

ramdev-mountlist

```
/*
 * Mountlist for manually mounting the sample ramdisk driver.
 *
 * F0: and F1: are set up for the V1.3 fast file system (FFS).
 * S2: and S3: are setup for the old file system (OFS).
 *
 * After mounting, the drives must be formatted.  Be sure to
 * use the FFS flag when formatting the Fast File System
 * ramdrives:
 *
 *      ;make sure "ramdev.device" is in DEVS:
 *
 *      mount f0: from mydev-mountlist
 *      format drive f0: name "Zippy" FFS
 */
F0:         Device = ramdev.device
            Unit   = 0
            LowCyl = 0 ; HighCyl = 14
            Surfaces  = 1
            Buffers = 1
            BlocksPerTrack = 10
            Flags  = 0
            Reserved = 2
            GlobVec = -1
            BufMemType = 0
            DosType = 0x444F5301
            StackSize = 4000
            FileSystem = l:fastfilesystem
#
F1:         Device = ramdev.device
            Unit   = 1
            LowCyl = 0 ; HighCyl = 14
            Surfaces  = 1
            Buffers = 1
            BlocksPerTrack = 10
            Flags  = 0
            Reserved = 2
            GlobVec = -1
            BufMemType = 0
            DosType = 0x444F5301
            StackSize = 4000
            FileSystem = l:fastfilesystem
#
S2:         Device = ramdev.device
            Unit   = 2
            Flags  = 0
            Surfaces  = 1
            BlocksPerTrack = 10
            Reserved = 1
            Interleave = 0
            LowCyl = 0  ;  HighCyl = 14
            Buffers = 1
            BufMemType = 0
#
S3:         Device = ramdev.device
            Unit   = 3
            Flags  = 0
            Surfaces  = 1
            BlocksPerTrack = 10
            Reserved = 1
            Interleave = 0
            LowCyl = 0  ;  HighCyl = 14
            Buffers = 1
            BufMemType = 0
#
```

ramdev.i

```
******************************************************************
*
*
* Copyright (C) 1986, Commodore Amiga Inc.  All rights reserved.
* Permission granted for non-commercial use
*
******************************************************************
*
* ramdev.i -- external declarations for skeleton ramdisk device
*
******************************************************************


;--- Assemble-time options
INFO_LEVEL  EQU 0      ; Specify amount of debugging info desired
                       ; If > 0 you must link with debug.lib!
                       ; You will need to run a terminal program to
                       ; set the baud rate.
*INTRRUPT   SET 1      ; Remove "*" to enable fake interrupt code
AUTOMOUNT   EQU 0      ; Work with the "mount" command if 0
                       ; Do it automatically if 1

;--- stack size and priority for the process we will create
MYPROCSTACKSIZE   EQU   $900
MYPROCPRI         EQU   0   ;Devices are often 5, NOT higher

;--- Base constants
NUMBEROFTRACKS EQU  40  ;<<<< Change THIS to change size of ramdisk <<<<
SECTOR         EQU  512 ;# bytes per sector
SECSHIFT       EQU  9   ;Shift count to convert byte # to sector #
SECTORSPER     EQU  10  ;# Sectors per "track"

RAMSIZE        EQU  SECTOR*NUMBEROFTRACKS*SECTORSPER
                    ; Use this much RAM per unit
BYTESPERTRACK EQU   SECTORSPER*SECTOR

IAMPULLING     EQU  7     ; "I am pulling the interrupt" bit of INTCRL1
INTENABLE      EQU  4     ; "Interrupt Enable" bit of INTCRL2
INTCTRL1       EQU  $40   ; Interrupt control register offset on board
INTCTRL2       EQU  $42   ; Interrupt control register offset on board
INTACK         EQU  $50   ; My board's interrupt reset address
;----------------------------------------------------------------------
;
; device command definitions (copied from devices/trackdisk.i)
;
;----------------------------------------------------------------------
   BITDEF  TD,EXTCOM,15        ; for "extended" commands !!!

   DEVINIT
   DEVCMD    CMD_MOTOR         ; control the disk's motor (NO-OP)
   DEVCMD    CMD_SEEK          ; explicit seek (NO-OP)
   DEVCMD    CMD_FORMAT        ; format disk - equated to WRITE for RAMDISK
   DEVCMD    CMD_REMOVE        ; notify when disk changes (NO-OP)
   DEVCMD    CMD_CHANGENUM     ; number of disk changes (always 0)
   DEVCMD    CMD_CHANGESTATE   ; is there a disk in the drive? (always TRUE)
   DEVCMD    CMD_PROTSTATUS    ; is the disk write protected? (always FALSE)
   DEVCMD    CMD_RAWREAD       ; Not supported
   DEVCMD    CMD_RAWWRITE      ; Not supported
   DEVCMD    CMD_GETDRIVETYPE  ; Get drive type
   DEVCMD    CMD_GETNUMTRACKS  ; Get number of tracks
   DEVCMD    CMD_ADDCHANGEINT  ; Add disk change interrupt (NO-OP)
   DEVCMD    CMD_REMCHANGEINT  ; Remove disk change interrupt ( NO-OP)
   DEVCMD    MYDEV_END         ; place marker -- first illegal command #

DRIVE3_5        EQU     1
DRIVE5_25       EQU     2
```

ramdev.i

```
;-----------------------------------------------------------------------
;
; Layout of parameter packet for MakeDosNode
;
;-----------------------------------------------------------------------

    STRUCTURE MkDosNodePkt,0
   APTR    mdn_dosName    ; Pointer to DOS file handler name
   APTR    mdn_execName    ; Pointer to device driver name
   ULONG   mdn_unit    ; Unit number
   ULONG   mdn_flags    ; OpenDevice flags
   ULONG   mdn_tableSize    ; Environment size
   ULONG   mdn_sizeBlock    ; # longwords in a block
   ULONG   mdn_secOrg    ; sector origin -- unused
   ULONG   mdn_numHeads    ; number of surfaces
   ULONG   mdn_secsPerBlk    ; secs per logical block -- unused
   ULONG   mdn_blkTrack    ; secs per track
   ULONG   mdn_resBlks    ; reserved blocks -- MUST be at least 1!
   ULONG   mdn_prefac    ; unused
   ULONG   mdn_interleave    ; interleave
   ULONG   mdn_lowCyl    ; lower cylinder
   ULONG   mdn_upperCyl    ; upper cylinder
   ULONG   mdn_numBuffers    ; number of buffers
   ULONG   mdn_memBufType    ; Type of memory for AmigaDOS buffers
   STRUCT  mdn_dName,5    ; DOS file handler name "RAM0"
   LABEL   mdn_Sizeof    ; Size of this structure

;-----------------------------------------------------------------------
;
; device data structures
;
;-----------------------------------------------------------------------
; maximum number of units in this device
MD_NUMUNITS    EQU    4

    STRUCTURE MyDev,LIB_SIZE
   UBYTE   md_Flags
   UBYTE   md_Pad1
   ;now longword aligned
   ULONG   md_SysLib
   ULONG   md_SegList
   ULONG   md_Base       ; Base address of this device's expansion board
   STRUCT  md_Units,MD_NUMUNITS*4
   LABEL   MyDev_Sizeof

   STRUCTURE MyDevUnit,UNIT_SIZE    ;Odd # longwords
   UBYTE    mdu_UnitNum
   UBYTE    mdu_SigBit        ; Signal bit allocated for interrupts
   ;Now longword aligned!
   APTR     mdu_Device
   STRUCT   mdu_stack,MYPROCSTACKSIZE
   STRUCT   mdu_tcb,TC_SIZE       ; Task Control Block (TCB) for disk task
   ULONG    mdu_SigMask           ; Signal these bits on interrupt
   IFD    INTRRUPT
    STRUCT    mdu_is,IS_SIZE        ; Interrupt structure
    UWORD     mdu_pad1             ;Longword align
   ENDC
   STRUCT   mdu_RAM,RAMSIZE       ; RAM used to simulate disk
   LABEL    MyDevUnit_Sizeof

   ;------ state bit for unit stopped
   BITDEF   MDU,STOPPED,2

MYDEVNAME    MACRO
      DC.B    'ramdev.device',0
      ENDM
```

asmsupp.i

```
*************************************************************************
*
*       Copyright (C) 1985, Commodore Amiga Inc.  All rights reserved.
*       Permission granted for non-commercial use
*
* asmsupp.i -- random low level assembly support routines
*              used by the Commodore sample Library & Device
*
*************************************************************************
CLEAR   MACRO           ;quick way to clear a D register on 68000
        MOVEQ   #0,\1
        ENDM

;BHS     MACRO
;        BCC.\0  \1 ;\0 is the extension used on the macro (such as ".s")
;        ENDM
;BLO     MACRO
;        BCS.\0  \1
;        ENDM
;EVEN    MACRO           ; word align code stream
;        DS.W    0
;        ENDM

LINKSYS MACRO           ; link to a library without having to see a _LVO
        MOVE.L  A6,-(SP)
        MOVE.L  \2,A6
        JSR     _LVO\1(A6)
        MOVE.L  (SP)+,A6
        ENDM

CALLSYS MACRO           ; call a library via A6 without having to see _LVO
        JSR     _LVO\1(A6)
        ENDM

XLIB    MACRO           ; define a library reference without the _LVO
        XREF    _LVO\1
        ENDM
;
; Put a message to the serial port at 9600 baud.  Used as so:
;
;     PUTMSG   30,<'%s/Init: called'>
;
; Parameters can be printed out by pushing them on the stack and
; adding the appropriate C printf-style % formatting commands.
;
                XREF    KPutFmt
PUTMSG:         MACRO   * level,msg

                IFGE    INFO_LEVEL-\1

                PEA     subSysName(PC)
                MOVEM.L A0/A1/D0/D1,-(SP)
                LEA     msg\@(pc),A0    ;Point to static format string
                LEA     4*4(SP),A1      ;Point to args
                JSR     KPutFmt
                MOVEM.L (SP)+,D0/D1/A0/A1
                ADDQ.L  #4,SP
                BRA.S   end\@

msg\@           DC.B    \2
                DC.B    10
                DC.B    0
                DS.W    0
end\@
                ENDC
                ENDM
```

ramdev.device.asm Page 1

```
*************************************************************************
*
*   Copyright (C) 1986,1988,1989 Commodore Amiga Inc.  All rights reserved.
*   Permission granted for non-commercial use.
*
*************************************************************************
*
* ramdev.asm -- Skeleton device code.
*
* A sample 4 unit ramdisk that can be bound to an expansion slot device,
* or used without.  Works with the Fast File System.
* This code is required reading for device driver writers.  It contains
* information not found elsewhere.  This code is somewhat old; you probably
* don't want to copy it directly.
*
* This example includes a task, though a task is not actually needed for
* a simple ram disk.  Unlike a single set of hardware registers that
* may need to be shared by multiple tasks, ram can be freely shared.
* This example does not show arbitration of hardware resources.
*
* Tested with CAPE and Metacomco
*
*         Based on mydev.asm
*         10/07/86 Modified by Lee Erickson to be a simple disk device
*                  using RAM to simulate a disk.
*         02/02/88 Modified by C. Scheppner, renamed ramdev
*         09/28/88 Repaired by Bryce Nesbitt for new release
*         11/02/88 More clarifications
*         02/01/89 Even more clarifications & warnings
*         02/22/89 START/STOP fix from Marco Papa
*
* Bugs: If RTF_AUTOINIT fails, library base still left in memory.
*
*************************************************************************

   SECTION firstsection

   NOLIST
   include "exec/types.i"
   include "exec/devices.i"
   include "exec/initializers.i"
   include "exec/memory.i"
   include "exec/resident.i"
   include "exec/io.i"
   include "exec/ables.i"
   include "exec/errors.i"
   include "exec/tasks.i"
   include "hardware/intbits.i"

   include "asmsupp.i"  ;standard asmsupp.i, same as used for library
   include "ramdev.i"

   IFNE AUTOMOUNT
   include "libraries/expansion.i"
   include "libraries/configvars.i"
   include "libraries/configregs.i"
   ENDC
   LIST


ABSEXECBASE equ 4   ;Absolute location of the pointer to exec.library base


   ;------ These don't have to be external, but it helps some
   ;------ debuggers to have them globally visible
```

ramdev.device.asm Page 2

```
    XDEF   Init
    XDEF   Open
    XDEF   Close
    XDEF   Expunge
    XDEF   Null
    XDEF   myName
    XDEF   BeginIO
    XDEF   AbortIO

    ;Pull these _LVOs in from amiga.lib
    XLIB   AddIntServer
    XLIB   RemIntServer
    XLIB   Debug
    XLIB   InitStruct
    XLIB   OpenLibrary
    XLIB   CloseLibrary
    XLIB   Alert
    XLIB   FreeMem
    XLIB   Remove
    XLIB   AddPort
    XLIB   AllocMem
    XLIB   AddTask
    XLIB   PutMsg
    XLIB   RemTask
    XLIB   ReplyMsg
    XLIB   Signal
    XLIB   GetMsg
    XLIB   Wait
    XLIB   WaitPort
    XLIB   AllocSignal
    XLIB   SetTaskPri
    XLIB   GetCurrentBinding     ;Use to get list of boards for this driver
    XLIB   MakeDosNode
    XLIB   AddDosNode
    XLIB   CopyMemQuick  ;Highly optimized copy function from exec.library

    INT_ABLES        ;Macro from exec/ables.i


;-----------------------------------------------------------------------
; The first executable location.  This should return an error
; in case someone tried to run you as a program (instead of
; loading you as a device).

FirstAddress:
               moveq   #-1,d0
               rts

;-----------------------------------------------------------------------
; A romtag structure.  After your driver is brought in from disk, the
; disk image will be scanned for this structure to discover magic constants
; about you (such as where to start running you from...).
;-----------------------------------------------------------------------

   ; Most people will not need a priority and should leave it at zero.
   ; the RT_PRI field is used for configuring the roms.  Use "mods" from
   ; wack to look at the other romtags in the system
MYPRI   EQU   0

initDDescrip:
                                 ;STRUCTURE RT,0
     DC.W    RTC_MATCHWORD       ; UWORD RT_MATCHWORD (Magic cookie)
     DC.L    initDDescrip        ; APTR  RT_MATCHTAG  (Back pointer)
     DC.L    EndCode             ; APTR  RT_ENDSKIP   (To end of this hunk)
     DC.B    RTF_AUTOINIT        ; UBYTE RT_FLAGS     (magic-see "Init:")
     DC.B    VERSION             ; UBYTE RT_VERSION
     DC.B     NT_DEVICE           ; UBYTE RT_TYPE      (must be correct)
     DC.B     MYPRI               ; BYTE  RT_PRI
     DC.L     myName              ; APTR  RT_NAME      (exec name)
     DC.L     idString            ; APTR  RT_IDSTRING  (text string)
     DC.L     Init                ; APTR  RT_INIT
                 ; LABEL RT_SIZE


   ;This name for debugging use
   IFNE INFO_LEVEL  ;If any debugging enabled at all
subSysName:
    dc.b     "ramdev",0
   ENDC

   ; this is the name that the device will have
myName:      MYDEVNAME

 IFNE  AUTOMOUNT
ExLibName     dc.b 'expansion.library',0    ; Expansion Library Name
 ENDC

   ; a major version number.
VERSION:     EQU   37

   ; A particular revision.  This should uniquely identify the bits in the
   ; device.  I use a script that advances the revision number each time
   ; I recompile.  That way there is never a question of which device
   ; that really is.
REVISION:    EQU   1

   ; this is an identifier tag to help in supporting the device
   ; format is 'name version.revision (d.m.yy)',<cr>,<lf>,<null>
idString:    dc.b    'ramdev 37.1 (28.8.91)',13,10,0

   ; force word alignment
   ds.w    0


   ; The romtag specified that we were "RTF_AUTOINIT".  This means
   ; that the RT_INIT structure member points to one of these
   ; tables below.  If the AUTOINIT bit was not set then RT_INIT
   ; would point to a routine to run.

Init:
   DC.L    MyDev_Sizeof      ; data space size
   DC.L    funcTable         ; pointer to function initializers
   DC.L    dataTable         ; pointer to data initializers
   DC.L    initRoutine       ; routine to run


funcTable:
   ;------ standard system routines
   dc.l    Open
   dc.l    Close
   dc.l    Expunge
   dc.l    Null        ;Reserved for future use!

   ;------ my device definitions
   dc.l    BeginIO
   dc.l    AbortIO

   ;------ custom extended functions
   dc.l    FunctionA
   dc.l    FunctionB

   ;------ function table end marker
```

```
   dc.l   -1


   ;The data table initializes static data structures. The format is
   ;specified in exec/InitStruct routine's manual pages.  The
   ;INITBYTE/INITWORD/INITLONG macros are in the file "exec/initializers.i".
   ;The first argument is the offset from the device base for this
   ;byte/word/long. The second argument is the value to put in that cell.
   ;The table is null terminated
   ;
dataTable:
   INITBYTE    LN_TYPE,NT_DEVICE       ;Must be LN_TYPE!
   INITLONG    LN_NAME,myName
   INITBYTE    LIB_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
   INITWORD    LIB_VERSION,VERSION
   INITWORD    LIB_REVISION,REVISION
   INITLONG    LIB_IDSTRING,idString
   DC.W    0   ;terminate list


;-------- initRoutine -------------------------------------------------------
;
; FOR RTF_AUTOINIT:
;   This routine gets called after the device has been allocated.
;   The device pointer is in D0.  The AmigaDOS segment list is in a0.
;   If it returns the device pointer, then the device will be linked
;   into the device list.  If it returns NULL, then the device
;   will be unloaded.
;
; IMPORTANT:
;   If you don't use the "RTF_AUTOINIT" feature, there is an additional
;   caveat.  If you allocate memory in your Open function, remember that
;   allocating memory can cause an Expunge... including an expunge of your
;   device.  This must not be fatal.  The easy solution is don't add your
;   device to the list until after it is ready for action.
;
; This call is single-threaded by exec; please read the description for
; "Open" below.
;
; Register Usage
; ==============
; a3 -- Points to temporary RAM
; a4 -- Expansion library base
; a5 -- device pointer
; a6 -- Exec base
;----------------------------------------------------------------------
initRoutine:
   ;------ get the device pointer into a convenient A register
   PUTMSG   5,<'%s/Init: called'>
   movem.l  d1-d7/a0-a5,-(sp)    ; Preserve ALL modified registers
   move.l   d0,a5

   ;------ save a pointer to exec
   move.l   a6,md_SysLib(a5)    ;faster access than move.l 4,a6

   ;------ save pointer to our loaded code (the SegList)
   move.l   a0,md_SegList(a5)

 IFNE  AUTOMOUNT
************************************************************************
*
* Here starts the AutoConfig stuff.  If this driver was to be tied to
* an expansion board, you would put this driver in the expansion drawer,
* and be called when BindDrivers finds a board that matches this driver.
* The Commodore-Amiga assigned product number of your board must be
* specified in the "PRODUCT=" field in the TOOLTYPES of this driver's icon.
```

ramdev.device.asm Page 5

```
* GetCurrentBinding() returns your (first) board.
*
   lea.l     ExLibName,A1       ; Get expansion lib. name
   moveq.l   #0,D0
   CALLSYS   OpenLibrary        ; Open the expansion library
   tst.l     D0
   beq       Init_Error

   ;------ init_OpSuccess:
   move.l    D0,A4          ;[expansionbase to A4]
   moveq     #0,D3
   lea       md_Base(A5),A0   ; Get the Current Bindings
   moveq     #4,D0            ; Just get address (length = 4 bytes)
   LINKLIB   _LVOGetCurrentBinding,A4
   move.l    md_Base(A5),D0      ; Get start of list
   tst.l     D0             ; If controller not found
   beq       Init_End       ; Exit and unload driver

   PUTMSG    10,<'%s/Init: GetCurrentBinding returned non-zero'>
   move.l    D0,A0            ; Get config structure address
   move.l    cd_BoardAddr(A0),md_Base(A5); Save board base address
   bclr.b    #CDB_CONFIGME,cd_Flags(A0); Mark board as configured

;----------------------------------------------------------------------
;
; Here we build a packet describing the characteristics of our disk to
; pass to AmigaDOS.  This serves the same purpose as a "mount" command
; of this device would.  For disks, it might be useful to actually
; get this information right from the disk itself.  Just as mount,
; it could be for multiple partitions on the single physical device.
; For this example, we will simply hard code the appropriate parameters.
;
; The AddDosNode call adds things to dos's list without needing to
; use mount.  We'll mount all 4 of our units whenever we are
; started.
;
;----------------------------------------------------------------------

;!!! If your card was successfully configured, you can mount the
;!!! units as DOS nodes

   ;------   Allocate temporary RAM to build MakeDosNode parameter packet
   move.l    #MEMF_CLEAR!MEMF_PUBLIC,d1
   move.l    #mdn_Sizeof,d0    ; Enough room for our parameter packet
   CALLSYS   AllocMem
   move.l    d0,a3           ;:BUG: AllocMem error not checked here.

   ;-----   Use InitStruct to initialize the constant portion of packet
   move.l    d0,a2          ; Point to memory to initialize
   moveq.l   #0,d0          ; Don't need to re-zero it
   lea.l     mdn_Init(pc),A1
   CALLSYS   InitStruct

   lea       mdn_dName(a3),a0     ; Get addr of Device name
   move.l    a0,mdn_dosName(a3)   ;   and save in environment

   moveq     #0,d6             ; Now tell AmigaDOS about all units UNITNUM
Uloop:
   move.b    d6,d0             ; Get unit number
   add.b     #$30,d0          ;; Make ASCII, minus 1
   move.b    d0,mdn_dName+2(a3)   ;   and store in name
   move.l    d6,mdn_unit(a3)      ; Store unit # in environment

;
;! Before adding to the dos list, you should really check if you
;! are about to cause a name collision.  This example does not.
```

ramdev.device.asm Page 6

```
;

   move.l    a3,a0
   LINKLIB   _LVOMakeDosNode,a4    ; Build AmigaDOS structures
   ;This can fail, but so what?
   move.l    d0,a0                 ; Get deviceNode address
   moveq.l   #0,d0                 ; Set device priority to 0
   moveq.l   #0,d1
*  moveq.l   #ADNF_STARTPROC,d1      ; See note below
   ;It's ok to pass a zero in here
   LINKLIB   _LVOAddDosNode,a4


; ADNF_STARTPROC will work, but only if dn_SegList is filled in
; in the SegPtr of the handler task.


   addq      #1,d6           ; Bump unit number
   cmp.b     #MD_NUMUNITS,d6
   bls.s     Uloop           ; Loop until all units installed

   move.l    a3,a1       ; Return RAM to system
   move.l    #mdn_Sizeof,d0
   CALLSYS   FreeMem

Init_End:

   move.l    a4,a1       ; Now close expansion library
   CALLSYS   CloseLibrary
*
*   You would normally set d0 to a NULL if your initialization failed,
*   but I'm not doing that for this demo, since it is unlikely
*   you actually have a board with any particular manufacturer ID
*   installed when running this demo.
*************************************************************************
  ENDC

  move.l   a5,d0
Init_Error:
  movem.l  (sp)+,d1-d7/a0-a5
  rts


;----------------------------------------------------------------------
;
; Here begins the system interface commands.  When the user calls
; OpenDevice/CloseDevice/RemDevice, this eventually gets translated
; into a call to the following routines (Open/Close/Expunge).
; Exec has already put our device pointer in a6 for us.
;
; IMPORTANT:
;   These calls are guaranteed to be single-threaded; only one task
;   will execute your Open/Close/Expunge at a time.
;
;   For Kickstart V33/34, the single-threading method involves "Forbid".
;   There is a good chance this will change.  Anything inside your
;   Open/Close/Expunge that causes a direct or indirect Wait() will break
;   the Forbid().  If the Forbid() is broken, some other task might
;   manage to enter your Open/Close/Expunge code at the same time.
;   Take care!
;
; Since exec has turned off task switching while in these routines
; (via Forbid/Permit), we should not take too long in them.
;
;----------------------------------------------------------------------
```

```
   ; Open sets the IO_ERROR field on an error.  If it was successfull,
   ; we should also set up the IO_UNIT and LN_TYPE fields.
   ; exec takes care of setting up IO_DEVICE.

Open:      ; ( device:a6, iob:a1, unitnum:d0, flags:d1 )

;** Subtle point: any AllocMem() call can cause a call to this device's
;** expunge vector.  If LIB_OPENCNT is zero, the device might get expunged.
   addq.w   #1,LIB_OPENCNT(a6)  ;Fake an opener for duration of call <|>

   PUTMSG   20,<'%s/Open: called'>
   movem.l  d2/a2/a3/a4,-(sp)

   move.l   a1,a2       ; save the iob

   ;------ see if the unit number is in range   *!* UNIT 0 to 3 *!*
   cmp.l    #MD_NUMUNITS,d0
   bcc.s    Open_Range_Error   ; unit number out of range (BHS)

   ;------ see if the unit is already initialized
   move.l   d0,d2       ; save unit number
   lsl.l    #2,d0
   lea.l    md_Units(a6,d0.l),a4
   move.l   (a4),d0
   bne.s    Open_UnitOK

   ;------ try and conjure up a unit
   bsr      InitUnit    ;scratch:a3 unitnum:d2 devpoint:a6

   ;------ see if it initialized OK
   move.l   (a4),d0
   beq.s    Open_Error

Open_UnitOK:
   move.l   d0,a3       ; unit pointer in a3
   move.l   d0,IO_UNIT(a2)

   ;------ mark us as having another opener
   addq.w   #1,LIB_OPENCNT(a6)
   addq.w   #1,UNIT_OPENCNT(a3)     ;Internal bookkeeping

   ;------ prevent delayed expunges
   bclr     #LIBB_DELEXP,md_Flags(a6)

   CLEAR    d0
   move.b   d0,IO_ERROR(a2)
   move.b   #NT_REPLYMSG,LN_TYPE(a2) ;IMPORTANT: Mark IORequest as "complete"

Open_End:

   subq.w   #1,LIB_OPENCNT(a6) ;** End of expunge protection <|>
   movem.l  (sp)+,d2/a2/a3/a4
   rts

Open_Range_Error:
Open_Error:
   moveq    #IOERR_OPENFAIL,d0
   move.b   d0,IO_ERROR(a2)
   move.l   d0,IO_DEVICE(a2)    ;IMPORTANT: trash IO_DEVICE on open failure
   PUTMSG   2,<'%s/Open: failed'>
   bra.s    Open_End


;----------------------------------------------------------------------------
; There are two different things that might be returned from the Close
; routine.  If the device wishes to be unloaded, then Close must return
```

```
; the segment list (as given to Init).  Otherwise close MUST return NULL.

Close:       ; ( device:a6, iob:a1 )
   movem.l  d1/a2-a3,-(sp)
   PUTMSG   20,<'%s/Close: called'>

   move.l   a1,a2

   move.l   IO_UNIT(a2),a3

   ;------ IMPORTANT: make sure the IORequest is not used again
   ;------ with a -1 in IO_DEVICE, any BeginIO() attempt will
   ;------ immediatly halt (which is better than a subtle corruption
   ;------ that will lead to hard-to-trace crashes!!!!!!!!!!!!!!!!!!
   moveq.l  #-1,d0
   move.l   d0,IO_UNIT(a2)      ;We're closed...
   move.l   d0,IO_DEVICE(a2)    ;customers not welcome at this IORequest!!

   ;------ see if the unit is still in use
   subq.w   #1,UNIT_OPENCNT(a3)

;!!!!!! Since this example is a RAM disk (and we don't want the contents to
;!!!!!! disappear between opens, ExpungeUnit will be skipped here.  It would
;!!!!!! be used for drivers of "real" devices
;!!!!!!   bne.s   Close_Device
;!!!!!!   bsr     ExpungeUnit

Close_Device:
   CLEAR    d0
   ;------ mark us as having one fewer openers
   subq.w   #1,LIB_OPENCNT(a6)

   ;------ see if there is anyone left with us open
   bne.s    Close_End

   ;------ see if we have a delayed expunge pending
   btst     #LIBB_DELEXP,md_Flags(a6)
   beq.s    Close_End

   ;------ do the expunge
   bsr      Expunge

Close_End:
   movem.l   (sp)+,d1/a2-a3
   rts                          ;MUST return either zero or the SegList!!!


;------- Expunge -----------------------------------------------------------
;
; Expunge is called by the memory allocator when the system is low on
; memory.
;
; There are two different things that might be returned from the Expunge
; routine.  If the device is no longer open then Expunge may return the
; segment list (as given to Init).  Otherwise Expunge may set the
; delayed expunge flag and return NULL.
;
; One other important note: because Expunge is called from the memory
; allocator, it may NEVER Wait() or otherwise take long time to complete.
;
;       A6          - library base (scratch)
;       D0-D1/A0-A1 - scratch
;
Expunge:   ; ( device: a6 )
   PUTMSG   10,<'%s/Expunge: called'>
   movem.l  d1/d2/a5/a6,-(sp)   ; Save ALL modified registers
   move.l   a6,a5
   move.l   md_SysLib(a5),a6

   ;------ see if anyone has us open
   tst.w   LIB_OPENCNT(a5)
;!!!!!  The following line is commented out for this RAM disk demo, since
;!!!!!  we don't want the RAM to be freed after FORMAT, for example.
;   beq    1$

   ;------ it is still open.  set the delayed expunge flag
   bset    #LIBB_DELEXP,md_Flags(a5)
   CLEAR   d0
   bra.s   Expunge_End

1$:
   ;------ go ahead and get rid of us.  Store our seglist in d2
   move.l   md_SegList(a5),d2

   ;------ unlink from device list
   move.l    a5,a1
   CALLSYS   Remove             ;Remove first (before FreeMem)

   ;
   ; device specific closings here...
   ;

   ;------ free our memory (must calculate from LIB_POSSIZE & LIB_NEGSIZE)
   move.l   a5,a1               ;Devicebase
   CLEAR    d0
   move.w   LIB_NEGSIZE(a5),d0
   suba.l   d0,a1               ;Calculate base of functions
   add.w    LIB_POSSIZE(a5),d0  ;Calculate size of functions + data area
   CALLSYS  FreeMem

   ;------ set up our return value
   move.l   d2,d0

Expunge_End:
   movem.l  (sp)+,d1/d2/a5/a6
   rts


;------- Null ---------------------------------------------------------------
Null:
   PUTMSG  1,<'%s/Null: called'>
   CLEAR   d0
   rts      ;The "Null" function MUST return NULL.


;------- Custom -------------------------------------------------------------
;
;Two "do nothing" device-specific functions
;
FunctionA:
    add.l   d1,d0    ;Add
    rts
FunctionB:
    add.l   d0,d0    ;Double
    rts


****************************************************************************

InitUnit:   ; ( d2:unit number, a3:scratch, a6:devptr )
   PUTMSG   30,<'%s/InitUnit: called'>
```

```
   movem.l  d2-d4/a2,-(sp)

   ;------ allocate unit memory
   move.l   #MyDevUnit_Sizeof,d0
   move.l   #MEMF_PUBLIC!MEMF_CLEAR,d1
   LINKSYS  AllocMem,md_SysLib(a6)
   tst.l    d0
   beq      InitUnit_End
   move.l   d0,a3

   moveq.l  #0,d0           ; Don't need to re-zero it
   move.l   a3,a2           ; InitStruct is initializing the UNIT
   lea.l    mdu_Init(pc),A1
   LINKSYS  InitStruct,md_SysLib(a6)

   ;!! IMPORTANT !!
   move.l   #42414400,mdu_RAM(a3)   ;Mark offset zero as ASCII "BAD "
   ;!! IMPORTANT !!

   move.b   d2,mdu_UnitNum(a3)      ;initialize unit number
   move.l   a6,mdu_Device(a3)       ;initialize device pointer

   ;------ start up the unit task.  We do a trick here --
   ;------ we set his message port to PA_IGNORE until the
   ;------ new task has a change to set it up.
   ;------ We cannot go to sleep here: it would be very nasty
   ;------ if someone else tried to open the unit
   ;------ (exec's OpenDevice has done a Forbid() for us --
   ;------ we depend on this to become single threaded).

   ;------ Initialize the stack information
   lea      mdu_stack(a3),a0          ; Low end of stack
   move.l   a0,mdu_tcb+TC_SPLOWER(a3)
   lea      MYPROCSTACKSIZE(a0),a0    ; High end of stack
   move.l   a0,mdu_tcb+TC_SPUPPER(a3)
   move.l   a3,-(A0)                  ; argument -- unit ptr (send on stack)
   move.l   a0,mdu_tcb+TC_SPREG(a3)
   lea      mdu_tcb(a3),a0
   move.l   a0,MP_SIGTASK(a3)

   IFGE INFO_LEVEL-30
       move.l   a0,-(SP)
       move.l   a3,-(SP)
       PUTMSG   30,<'%s/InitUnit, unit= %lx, task=%lx'>
       addq.l   #8,sp
   ENDC

   ;------ initialize the unit's message port's list
   lea      MP_MSGLIST(a3),a0
   NEWLIST  a0                    ;<- IMPORTANT! Lists MUST! have NEWLIST
                                  ;work magic on them before use.  (AddPort()
                                  ;can do this for you)

   IFD   INTRRUPT
   move.l   a3,mdu_is+IS_DATA(a3)   ; Pass unit addr to interrupt server
   ENDC

;   Startup the task
   lea      mdu_tcb(a3),a1
   lea      Task_Begin(PC),a2
   move.l   a3,-(sp)       ; Preserve UNIT pointer
   lea      -1,a3          ; generate address error
                           ; if task ever "returns" (we RemTask() it
                           ; to get rid of it...)
   CLEAR   d0
   PUTMSG   30,<'%s/About to add task'>
   LINKSYS AddTask,md_SysLib(a6)
   move.l   (sp)+,a3       ; restore UNIT pointer

   ;------ mark us as ready to go
   move.l   d2,d0          ; unit number
   lsl.l    #2,d0
   move.l   a3,md_Units(a6,d0.l)   ; set unit table
   PUTMSG   30,<'%s/InitUnit: ok'>

InitUnit_End:
   movem.l   (sp)+,d2-d4/a2
   rts


;----------------------------------------------------------------------------
FreeUnit:   ; ( a3:unitptr, a6:deviceptr )
   move.l   a3,a1
   move.l   #MyDevUnit_Sizeof,d0
   LINKSYS  FreeMem,md_SysLib(a6)
   rts

;----------------------------------------------------------------------------
ExpungeUnit:    ; ( a3:unitptr, a6:deviceptr )
   PUTMSG   10,<'%s/ExpungeUnit: called'>
   move.l   d2,-(sp)

;
; If you can expunge you unit, and each unit has it's own interrupts,
; you must remember to remove its interrupt server
;

   IFD    INTRRUPT
   lea.l   mdu_is(a3),a1              ; Point to interrupt structure
   moveq   #INTB_PORTS,d0             ; Portia interrupt bit 3
   LINKSYS RemIntServer,md_SysLib(a6) ;Now remove the interrupt server
   ENDC

   ;------ get rid of the unit's task.  We know this is safe
   ;------ because the unit has an open count of zero, so it
   ;------ is 'guaranteed' not in use.
   lea    mdu_tcb(a3),a1
   LINKSYS RemTask,md_SysLib(a6)

   ;------ save the unit number
   CLEAR   d2
   move.b  mdu_UnitNum(a3),d2

   ;------ free the unit structure.
   bsr     FreeUnit

   ;------ clear out the unit vector in the device
   lsl.l   #2,d2
   clr.l   md_Units(a6,d2.l)

   move.l  (sp)+,d2
   rts


*****************************************************************************
;
; here begins the device functions
;
;----------------------------------------------------------------------------
; cmdtable is used to look up the address of a routine that will
; implement the device command.
```

```
;
; NOTE: the "extended" commands (ETD_READ/ETD_WRITE) have bit 15 set!
; We deliberately refuse to operate on such commands.  However a driver
; that supports removable media may want to implement this.  One
; open issue is the handling of the "seclabel" area. It is probably
; best to reject any command with a non-null "seclabel" pointer.
;
cmdtable:
   DC.L   Invalid      ;$00000001  ;0  CMD_INVALID
   DC.L   MyReset      ;$00000002  ;1  CMD_RESET
   DC.L   RdWrt        ;$00000004  ;2  CMD_READ         (\/common)
   DC.L   RdWrt        ;$00000008  ;3  CMD_WRITE        (/\common)  ETD_
   DC.L   Update       ;$00000010  ;4  CMD_UPDATE       (NO-OP)     ETD_
   DC.L   Clear        ;$00000020  ;5  CMD_CLEAR        (NO-OP)     ETD_
   DC.L   MyStop       ;$00000040  ;6  CMD_STOP                     ETD_
   DC.L   Start        ;$00000080  ;7  CMD_START
   DC.L   Flush        ;$00000100  ;8  CMD_FLUSH
   DC.L   Motor        ;$00000200  ;9  TD_MOTOR         (NO-OP)     ETD_
   DC.L   Seek         ;$00000400  ;A  TD_SEEK          (NO-OP)     ETD_
   DC.L   RdWrt        ;$00000800  ;B  TD_FORMAT        (Same as write)
   DC.L   MyRemove     ;$00001000  ;C  TD_REMOVE        (NO-OP)
   DC.L   ChangeNum    ;$00002000  ;D  TD_CHANGENUM     (returns 0)
   DC.L   ChangeState  ;$00004000  ;E  TD_CHANGESTATE   (returns 0)
   DC.L   ProtStatus   ;$00008000  ;F  TD_PROTSTATUS    (returns 0)
   DC.L   RawRead      ;$00010000  ;10 TD_RAWREAD       (INVALID)
   DC.L   RawWrite     ;$00020000  ;11 TD_RAWWRITE      (INVALID)
   DC.L   GetDriveType ;$00040000  ;12 TD_GETDRIVETYPE  (Returns 1)
   DC.L   GetNumTracks ;$00080000  ;13 TD_GETNUMTRACKS  (Returns NUMTRKS)
   DC.L   AddChangeInt ;$00100000  ;14 TD_ADDCHANGEINT  (NO-OP)
   DC.L   RemChangeInt ;$00200000  ;15 TD_REMCHANGEINT  (NO-OP)
cmdtable_end:

; this define is used to tell which commands should be handled
; immediately (on the caller's schedule).
;
; The immediate commands are Invalid, Reset, Stop, Start, Flush
;
; Note that this method limits you to just 32 device specific commands,
; which may not be enough.
;IMMEDIATES   EQU   %00000000000000000000000111000011
;;                   --------================--------
;;                   FEDCBA9876543210FEDCBA9876543210

;;An alternate version.  All commands that are trivially short
;;and %100 reentrant are included.  This way you won't get the
;;task switch overhead for these commands.
;;
IMMEDIATES   EQU   %11111111111111111111011111110011
;                   --------================--------
;                   FEDCBA9876543210FEDCBA9876543210

    IFD   INTRRUPT   ; if using interrupts,
; These commands can NEVER be done "immediately" if using interrupts,
; since they would "wait" for the interrupt forever!
; Read, Write, Format
NEVERIMMED   EQU   $0000080C
    ENDC


;--------------------------------
; BeginIO starts all incoming io.  The IO is either queued up for the
; unit task or processed immediately.
;
;
; BeginIO often is given the responsibility of making devices single
; threaded... so two tasks sending commands at the same time don't cause
```

ramdev.device.asm Page 13

```
; a problem.  Once this has been done, the command is dispatched via
; PerformIO.
;
; There are many ways to do the threading.  This example uses the
; UNITB_ACTIVE bit.  Be sure this is good enough for your device before
; using!  Any method is ok.  If immediate access can not be obtained, the
; request is queued for later processing.
;
; Some IO requests do not need single threading, these can be performed
; immediatley.
;
; IMPORTANT:
;   The exec WaitIO() function uses the IORequest node type (LN_TYPE)
;   as a flag.  If set to NT_MESSAGE, it assumes the request is
;   still pending and will wait.  If set to NT_REPLYMSG, it assumes the
;   request is finished.  It's the responsibility of the device driver
;   to set the node type to NT_MESSAGE before returning to the user.
;
BeginIO:    ; ( iob: a1, device:a6 )

    IFGE INFO_LEVEL-1
        bchg.b  #1,$bfe001  ;Blink the power LED
    ENDC
    IFGE INFO_LEVEL-3
     clr.l     -(sp)
     move.w    IO_COMMAND(a1),2(sp)  ;Get entire word
     PUTMSG    3,<'%s/BeginIO  -- $%lx'>
     addq.l    #4,sp
    ENDC

    movem.l   d1/a0/a3,-(sp)

    move.b  #NT_MESSAGE,LN_TYPE(a1) ;So WaitIO() is guaranteed to work
    move.l  IO_UNIT(a1),a3          ;bookkeeping -> what unit to play with
    move.w  IO_COMMAND(a1),d0

    ;Do a range check & make sure ETD_XXX type requests are rejected
    cmp.w   #MYDEV_END,d0       ;Compare all 16 bits
    bcc     BeginIO_NoCmd       ;no, reject it.  (bcc=bhs - unsigned)

    ;------ process all immediate commands no matter what
    move.l  #IMMEDIATES,d1
    DISABLE a0                  ;<-- Ick, nasty stuff, but needed here.
    btst.l  d0,d1
    bne     BeginIO_Immediate

    IFD   INTRRUPT   ; if using interrupts,
     ;------ queue all NEVERIMMED commands no matter what
     move.w  #NEVERIMMED,d1
     btst    d0,d1
     bne.s   BeginIO_QueueMsg
    ENDC


    ;------ see if the unit is STOPPED.  If so, queue the msg.
    btst    #MDUB_STOPPED,UNIT_FLAGS(a3)
    bne     BeginIO_QueueMsg


    ;------ This is not an immediate command.  See if the device is
    ;------ busy.  If the device is not, do the command on the
    ;------ user schedule.  Else fire up the task.
    ;------ This type of arbitration is not really needed for a ram
    ;------ disk, but is essential for a device to reliably work
    ;------ with shared hardware
    ;------
```

ramdev.device.asm Page 14

```
    ;------ When the lines below are ";" commented out, the task gets
    ;------ a better workout.  When the lines are active, the calling
    ;------ process is usually used for the operation.
    ;------
    ;------ REMEMBER:::: Never Wait() on the user's schedule in BeginIO()!
    ;------ The only exception is when the user has indicated it is ok
    ;------ by setting the "quick" bit.  Since this device copies from
    ;------ ram that never needs to be waited for, this subtlely may not
    ;------ be clear.
    ;------
    bset    #UNITB_ACTIVE,UNIT_FLAGS(a3)   ;<---- comment out these
    beq.s   BeginIO_Immediate              ;<---- lines to test task.


    ;------ we need to queue the device.  mark us as needing
    ;------ task attention.  Clear the quick flag
BeginIO_QueueMsg:
    bset    #UNITB_INTASK,UNIT_FLAGS(a3)
    bclr    #IOB_QUICK,IO_FLAGS(a1)    ;We did NOT complete this quickly
    ENABLE  a0


    IFGE INFO_LEVEL-250
     move.l  a1,-(sp)
     move.l  a3,-(sp)
     PUTMSG  250,<'%s/PutMsg: Port=%lx Message=%lx'>
     addq.l  #8,sp
    ENDC

    move.l  a3,a0
    LINKSYS  PutMsg,md_SysLib(a6)   ;Port=a0, Message=a1
    bra.s   BeginIO_End
    ;----- return to caller before completing


    ;------ Do it on the schedule of the calling process
    ;------
BeginIO_Immediate:
    ENABLE  a0
    bsr.s   PerformIO

BeginIO_End:
    PUTMSG  200,<'%s/BeginIO_End'>
    movem.l (sp)+,d1/a0/a3
    rts

BeginIO_NoCmd:
    move.b  #IOERR_NOCMD,IO_ERROR(a1)
    bra.s   BeginIO_End


;
; PerformIO actually dispatches an io request.  It might be called from
; the task, or directly from BeginIO (thus on the callers's schedule)
;
; It expects a3 to already
; have the unit pointer in it.  a6 has the device pointer (as always).
; a1 has the io request.  Bounds checking has already been done on
; the I/O Request.
;

PerformIO:   ; ( iob:a1, unitptr:a3, devptr:a6 )
    IFGE INFO_LEVEL-150
     clr.l     -(sp)
     move.w    IO_COMMAND(a1),2(sp)  ;Get entire word
     PUTMSG    150,<'%s/PerformIO -- $%lx'>
     addq.l   #4,sp
    ENDC

    moveq   #0,d0
    move.b  d0,IO_ERROR(A1)       ; No error so far
    move.b  IO_COMMAND+1(a1),d0 ;Look only at low byte
    lsl.w   #2,d0                 ; Multiply by 4 to get table offset
    lea.l   cmdtable(pc),a0
    move.l  0(a0,d0.w),a0

    jmp     (a0)    ;iob:a1  unit:a3  devprt:a6



;
; TermIO sends the IO request back to the user.  It knows not to mark
; the device as inactive if this was an immediate request or if the
; request was started from the server task.
;

TermIO:       ; ( iob:a1, unitptr:a3, devptr:a6 )
    PUTMSG  160,<'%s/TermIO'>
    move.w  IO_COMMAND(a1),d0

    move.w  #IMMEDIATES,d1
    btst    d0,d1
    bne.s   TermIO_Immediate    ;IO was immediate, don't do task stuff...

    ;------ we may need to turn the active bit off.
    btst    #UNITB_INTASK,UNIT_FLAGS(a3)
    bne.s   TermIO_Immediate    ;IO was came from task, don't clear ACTIVE...

    ;------ the task does not have more work to do
    bclr    #UNITB_ACTIVE,UNIT_FLAGS(a3)

TermIO_Immediate:
    ;------ if the quick bit is still set then we don't need to reply
    ;------ msg -- just return to the user.
    btst    #IOB_QUICK,IO_FLAGS(a1)
    bne.s   TermIO_End
    LINKSYS ReplyMsg,md_SysLib(a6)       ;a1-message
    ;(ReplyMsg sets the LN_TYPE to NT_REPLYMSG)

TermIO_End:
    rts

*******************************************************************************
;
; Here begins the functions that implement the device commands
; all functions are called with:
;   a1 -- a pointer to the io request block
;   a3 -- a pointer to the unit
;   a6 -- a pointer to the device
;
; Commands that conflict with 68000 instructions have a "My" prepended
; to them.
;----------------------------------------------------------------------

;We can't AbortIO anything, so don't touch the IORequest!
;
;AbortIO() is a REQUEST to "hurry up" processing of an IORequest.
;If the IORequest was already complete, nothing happens (if an IORequest
;is quick or LN_TYPE=NT_REPLYMSG, the IORequest is complete).
;The message must be replied with ReplyMsg(), as normal.
;
```

```
AbortIO:        ; ( iob: a1, device:a6 )
    moveq   #IOERR_NOCMD,d0 ;return "AbortIO() request failed"
    rts

RawRead:        ; 10 Not supported   (INVALID)
RawWrite:       ; 11 Not supported   (INVALID)
Invalid:
    move.b  #IOERR_NOCMD,IO_ERROR(a1)
    bra.s   TermIO

;
; Update and Clear are internal buffering commands.  Update forces all
; io out to its final resting spot, and does not return until this is
; totally done.  Since this is automatic in a ramdisk, we simply return "Ok".
;
; Clear invalidates all internal buffers.  Since this device
; has no internal buffers, these commands do not apply.
;
Update:
Clear:
MyReset:                    ;Do nothing (nothing reasonable to do)
AddChangeInt:               ;Do nothing
RemChangeInt:               ;Do nothing
MyRemove:                   ;Do nothing
Seek:                       ;Do nothing
Motor:                      ;Do nothing
ChangeNum:                  ;Return zero (changecount =0)
ChangeState:                ;Zero indicates disk inserted
ProtStatus:                 ;Zero indicates unprotected
    clr.l   IO_ACTUAL(a1)
    bra.s   TermIO


GetDriveType:               ;make it look like 3.5" (90mm) drive
    moveq   #DRIVE3_5,d0
    move.l  d0,IO_ACTUAL(a1)
    bra.s   TermIO


GetNumTracks:
    move.l  #RAMSIZE/BYTESPERTRACK,IO_ACTUAL(a1) ;Number of tracks
    bra.s   TermIO

;
; Foo and Bar are two device specific commands that are provided just
; to show you how commands are added.  They currently return that
; no work was done.
;
Foo:
Bar:
    clr.l   IO_ACTUAL(a1)
    bra     TermIO


;----------------------------------------------------------------------
; This device is designed so that no combination of bad
; inputs can ever cause the device driver to crash.
;----------------------------------------------------------------------
RdWrt:
    IFGE INFO_LEVEL-200
        move.l  IO_DATA(a1),-(sp)
        move.l  IO_OFFSET(a1),-(sp)
        move.l  IO_LENGTH(a1),-(sp)
        PUTMSG  200,<'%s/RdWrt len %ld offset %ld data $%lx'>
        addq.l  #8,sp
        addq.l  #4,sp
        ENDC

        movem.l a2/a3,-(sp)
        move.l  a1,a2                   ;Copy iob
        move.l  IO_UNIT(a2),a3          ;Get unit pointer

*           check operation for legality
        btst.b  #0,IO_DATA+3(a2)        ;check if user's pointer is ODD
        bne.s   IO_LenErr               ;bad...
        ;[D0=offset]

        move.l  IO_OFFSET(a2),d0
        move.l  d0,d1
        and.l   #SECTOR-1,d1            ;Bad sector boundary or alignment?
        bne.s   IO_LenErr               ;bad...
        ;[D0=offset]

*           check for IO within disc range
        ;[D0=offset]
        add.l   IO_LENGTH(a2),d0        ;Add length to offset
        bcs.s   IO_LenErr               ;overflow... (important test)
        cmp.l   #RAMSIZE,d0             ;Last byte is highest acceptable total
        bhi.s   IO_LenErr               ;bad... (unsigned compare)
        and.l   #SECTOR-1,d0            ;Even sector boundary?
        bne.s   IO_LenErr               ;bad...

*           We've gotten this far, it must be a valid request.

        IFD   INTRRUPT
         move.l    mdu_SigMask(a3),d0  ; Get signals to wait for
         LINKSYS   Wait,md_SysLib(a6)  ; Wait for interrupt before proceeding
        ENDC


        lea.l   mdu_RAM(a3),a0          ; Point to RAMDISK "sector" for I/O
        add.l   IO_OFFSET(a2),a0        ; Add offset to ram base
        move.l  IO_LENGTH(a2),d0
        move.l  d0,IO_ACTUAL(a2)        ; Indicate we've moved all bytes
        beq.s   RdWrt_end               ;---deal with zero length I/O
        move.l  IO_DATA(a2),a1          ; Point to data buffer
;
;A0=ramdisk index
;A1=user buffer
;D0=length
;
        cmp.b   #CMD_READ,IO_COMMAND+1(a2)  ; Decide on direction
        BEQ.S   CopyTheBlock
        EXG     A0,A1                   ; For Write and Format, swap source & dest
CopyTheBlock:
        LINKSYS CopyMemQuick,md_SysLib(a6)  ;A0=source A1=dest D0=size
        ;CopyMemQuick is very fast

RdWrt_end:
        move.l  a2,a1
        movem.l (sp)+,a2/a3
        bra     TermIO          ;END



IO_LenErr:
        PUTMSG  10,<'bad length'>
        move.b  #IOERR_BADLENGTH,IO_ERROR(a2)
IO_End:
        clr.l   IO_ACTUAL(a2)           ;Initially, no data moved
        bra.s   RdWrt_end
```

```
;
; the Stop command stop all future io requests from being
; processed until a Start command is received.  The Stop
; command is NOT stackable: e.g. no matter how many stops
; have been issued, it only takes one Start to restart
; processing.
;
;Stop is rather silly for a ramdisk
MyStop:
   PUTMSG    30,<'%s/MyStop: called'>
   bset   #MDUB_STOPPED,UNIT_FLAGS(a3)
   bra    TermIO


Start:
    PUTMSG    30,<'%s/Start: called'>
    bsr.s  InternalStart
    bra    TermIO

                ;[A3=unit A6=device]
InternalStart:
    move.l  a1,-(sp)
    ;------ turn processing back on
    bclr    #MDUB_STOPPED,UNIT_FLAGS(a3)
    ;------ kick the task to start it moving
    move.b  MP_SIGBIT(a3),d1
    CLEAR   d0
    bset    d1,d0                       ;prepared signal mask
    move.l  MP_SIGTASK(a3),a1           ;:FIXED:marco-task to signal
    LINKSYS Signal,md_SysLib(a6)        ;:FIXED:marco-a6 not a3
    move.l  (sp)+,a1
    rts


;
; Flush pulls all I/O requests off the queue and sends them back.
; We must be careful not to destroy work in progress, and also
; that we do not let some io requests slip by.
;
; Some funny magic goes on with the STOPPED bit in here.  Stop is
; defined as not being reentrant.  We therefore save the old state
; of the bit and then restore it later.  This keeps us from
; needing to DISABLE in flush.  It also fails miserably if someone
; does a start in the middle of a flush. (A semaphore might help...)
;

Flush:
   PUTMSG    30,<'%s/Flush: called'>
   movem.l   d2/a1/a6,-(sp)

   move.l    md_SysLib(a6),a6

   bset    #MDUB_STOPPED,UNIT_FLAGS(a3)
   sne    d2

Flush_Loop:
   move.l    a3,a0
   CALLSYS   GetMsg      ;Steal messages from task's port

   tst.l    d0
   beq.s    Flush_End

   move.l    d0,a1
   move.b    #IOERR_ABORTED,IO_ERROR(a1)
   CALLSYS   ReplyMsg

   bra.s   Flush_Loop

Flush_End:
   move.l    d2,d0
   movem.l    (sp)+,d2/a1/a6

   tst.b   d0
   beq.s   1$

   bsr   InternalStart
1$:
   bra     TermIO

*******************************************************************************
;
; Here begins the task related routines
;
; A Task is provided so that queued requests may be processed at
; a later time.  This is not very justifiable for a ram disk, but
; is very useful for "real" hardware devices.  Take care with
; your arbitration of shared hardware with all the multitasking
; programs that might call you at once.
;
; Register Usage
; ==============
; a3 -- unit pointer
; a6 -- syslib pointer
; a5 -- device pointer
; a4 -- task (NOT process) pointer
; d7 -- wait mask
;----------------------------------------------------------------------

; some dos magic, useful for Processes (not us).  A process is started at
; the first  executable address  after a segment list.  We hand craft a
; segment list here.  See the the DOS technical reference if you really
; need to know more about this.
; The next instruction after the segment list is the first executable address

    cnop    0,4     ; long word align
    DC.L    16      ; segment length -- any number will do (this is 4
                    ; bytes back from the segment pointer)
myproc_seglist:
    DC.L    0       ; pointer to next segment

Task_Begin:
    PUTMSG  35,<'%s/Task_Begin'>
    move.l  ABSEXECBASE,a6

    ;------ Grab the argument passed down from our parent
    move.l  4(sp),a3             ; Unit pointer
    move.l  mdu_Device(a3),a5    ; Point to device structure

    IFD   INTRRUPT
     ;------ Allocate a signal for "I/O Complete" interrupts
     moveq   #-1,d0          ; -1 is any signal at all
     CALLSYS   AllocSignal
     move.b   d0,mdu_SigBit(A3)    ; Save in unit structure
     moveq   #0,d7           ; Convert bit number signal mask
     bset   d0,d7
     move.l   d7,mdu_SigMask(A3)    ; Save in unit structure
     lea.l   mdu_is(a3),a1        ; Point to interrupt structure
     moveq   #INTB_PORTS,d0       ; Portia interrupt bit 3
     CALLSYS AddIntServer         ; Now install the server
```

```
     move.l   md_Base(a5),a0       ; Get board base address
*    bset.b   #INTENABLE,INTCTRL2(a0)    ; Enable interrupts
    ENDC

    ;------ Allocate a signal
    moveq   #-1,d0          ; -1 is any signal at all
    CALLSYS AllocSignal
    move.b  d0,MP_SIGBIT(a3)
    move.b  #PA_SIGNAL,MP_FLAGS(a3) ;Make message port "live"
    ;------ change the bit number into a mask, and save in d7
    moveq   #0,d7        ;Clear D7
    bset    d0,d7

    IFGE INFO_LEVEL-40
     move.l  $114(a6),-(sp)
     move.l  a5,-(sp)
     move.l  a3,-(sp)
     move.l  d0,-(sp)
     PUTMSG  40,<'%s/Signal=%ld, Unit=%lx Device=%lx Task=%lx'>
     add.l   #4*4,sp
    ENDC

    bra.s   Task_StartHere

; OK, kids, we are done with initialization.  We now can start the main loop
; of the driver.  It goes like this.  Because we had the port marked
; PA_IGNORE for a while (in InitUnit) we jump to the getmsg code on entry.
; (The first message will probably be posted BEFORE our task gets a chance
; to run)
;------     wait for a message
;------     lock the device
;------     get a message.  If no message, unlock device and loop
;------     dispatch the message
;------     loop back to get a message

    ;------ no more messages.  back ourselves out.
Task_Unlock:
    and.b   #$ff&(~(UNITF_ACTIVE!UNITF_INTASK)),UNIT_FLAGS(a3)
    ;------ main loop: wait for a new message

Task_MainLoop:
    PUTMSG   75,<'%s/++Sleep'>
    move.l  d7,d0
    CALLSYS Wait
    IFGE INFO_LEVEL-5
       bchg.b  #1,$bfe001  ;Blink the power LED
    ENDC
Task_StartHere:
    PUTMSG   75,<'%s/++Wakeup'>
    ;------ see if we are stopped
    btst    #MDUB_STOPPED,UNIT_FLAGS(a3)
    bne.s   Task_MainLoop       ; device is stopped, ignore messages
    ;------ lock the device
    bset    #UNITB_ACTIVE,UNIT_FLAGS(a3)
    bne     Task_MainLoop       ; device in use (immediate command?)


   ;------ get the next request
Task_NextMessage:
    move.l  a3,a0
    CALLSYS GetMsg
    PUTMSG  1,<'%s/GotMsg'>
    tst.l   d0
    beq     Task_Unlock ; no message?

    ;------ do this request
    move.l  d0,a1
    exg     a5,a6       ; put device ptr in right place
    bsr     PerformIO
    exg     a5,a6       ; get syslib back in a6

    bra.s   Task_NextMessage

*******************************************************************************
;
; Here is a dummy interrupt handler, with some crucial components commented
; out.  If the IFD INTRRUPT is enabled, this code will cause the device to
; wait for a level two interrupt before it will process each request
; (pressing RETURN on the keyboard will do it).  This code is normally
; disabled, and must fake or omit certain operations since there  isn't
; really any hardware for this driver.  Similar code has been used
; successfully in other, "REAL" device drivers.
;

   IFD   INTRRUPT

;   A1 should be pointing to the unit structure upon entry! (IS_DATA)
myintr:
*       move.l   md_Base(a0),a0      ; point to board base address
*       btst.b   #IAMPULLING,INTCTRL1(a0);See if I'm interrupting
*       beq.s   myexnm          ; if not set, exit, not mine
*       move.b   #0,INTACK(a0)      ; toggle controller's int2 bit

;       ------ signal the task that an interrupt has occurred

        move.l  mdu_Device(a1),a0   ; Get device pointer
        move.l  mdu_SigMask(a1),d0
        lea.l   mdu_tcb(a1),a1
        move.l  md_SysLib(a0),a6   ; Get pointer to system
        CALLSYS Signal

;       now clear the zero condition code so that
;       the interrupt handler doesn't call the next
;       interrupt server.
;
*       moveq   #1,d0          clear zero flag
*       bra.s   myexit          now exit
;
;       this exit point sets the zero condition code
;       so the interrupt handler will try the next server
;       in the interrupt chain
;
myexnm      moveq   #0,d0         set zero condition code
;
myexit      rts
   ENDC


*******************************************************************************

mdu_Init:
;   ------ Initialize the device

    INITBYTE    MP_FLAGS,PA_IGNORE  ;Unit starts with a message port
    INITBYTE    LN_TYPE,NT_MSGPORT  ;
    INITLONG    LN_NAME,myName      ;
    INITLONG    mdu_tcb+LN_NAME,myName
    INITBYTE    mdu_tcb+LN_TYPE,NT_TASK
    INITBYTE    mdu_tcb+LN_PRI,5
    IFD   INTRRUPT
     INITBYTE    mdu_is+LN_PRI,4      ; Int priority 4
     INITLONG    mdu_is+IS_CODE,myintr  ; Interrupt routine addr
```

```
     INITLONG    mdu_is+LN_NAME,myName
    ENDC
    DC.W    0

 IFNE  AUTOMOUNT
mdn_Init:
*   ;------ Initialize packet for MakeDosNode

    INITLONG    mdn_execName,myName     ; Address of driver name
    INITLONG    mdn_tableSize,12        ; # long words in AmigaDOS env.
    INITLONG    mdn_dName,$524d0000     ; Store 'RM' in name
    INITLONG    mdn_sizeBlock,SECTOR/4  ; # longwords in a block
    INITLONG    mdn_numHeads,1          ; RAM disk has only one "head"
    INITLONG    mdn_secsPerBlk,1        ; secs/logical block, must = "1"
    INITLONG    mdn_blkTrack,SECTORSPER ; secs/track (must be reasonable)
    INITLONG    mdn_resBlks,1           ; reserved blocks, MUST > 0!
    INITLONG    mdn_upperCyl,(RAMSIZE/BYTESPERTRACK)-1 ; upper cylinder
    INITLONG    mdn_numBuffers,1        ; # AmigaDOS buffers to start
    DC.W    0
 ENDC

;----------------------------------------------------------------------------
; EndCode is a marker that shows the end of your code.  Make sure it does not
; span hunks, and is not before the rom tag!  It is ok to put it right after
; the rom tag -- that way you are always safe.  I put it here because it
; happens to be the "right" thing to do, and I know that it is safe in this
; case (this program has only a single code hunk).
;----------------------------------------------------------------------------
EndCode:    END
```

appendix C

FLOPPY BOOT PROCESS AND PHYSICAL LAYOUT

The first two sectors on each floppy disk contain special boot information. These sectors are read into the system at an arbitrary position; therefore, the code *must* be position independent. The first three longwords come from the include file *devices/bootblock.h.* The type must be BBID_DOS; the checksum must be correct (an additive carry wraparound sum of 0xffffffff). Execution starts at location 12 of the first sector read in.

The code is called with an open trackdisk.device I/O request pointer in A1 (see the "Trackdisk" chapter for more information). The boot code is free to use the IO request as it wishes (the code may trash A1, but must not trash the I/O request itself).

The boot code must return values in two registers: D0 and A0. D0 is a failure code – if it is non-zero then a system alert will be called, and the system will reboot.

If D0 is zero then A0 must contain the start address to jump to. The strap module will free the boot sector memory, free the boot picture memory, close the trackdisk.device I/O request, do any other cleanup that is required, then jump to the location pointed to by A0.

Boot code may allocate memory, use trackdisk.device to load relocatable information into the memory, then return with D0=0 and A0 pointing to code. The system will clean up, then call the code.

COMMODORE-AMIGA DISK FORMAT

The following are details about how the bits on the Commodore-Amiga disk are actually written.

```
Gross Data Organization:

    3 1/2 inch (90mm) disk
    double-sided
    80 cylinders/160 tracks

Per-track Organization:

    Nulls written as a gap, then 11 or 22 sectors of data.
    No gaps written between sectors.

Per-sector Organization:

    All data is MFM encoded.  This is the pre-encoded contents
    of each sector:

        two bytes of 00 data     (MFM = $AAAA each)
        two bytes of A1*         ("standard sync byte" -- MFM
                                  encoded A1 without a clock pulse)
                                 (MFM = $4489 each)
        one byte of format byte  (Amiga 1.0 format = $FF)
        one byte of track number
        one byte of sector number
        one byte of sectors until end of write (NOTE 1)
            [above 4 bytes treated as one longword
             for purposes of MFM encoding]
        16 bytes of OS recovery info (NOTE 2)
            [treated as a block of 16 bytes for encoding]
        four bytes of header checksum
            [treated as a longword for encoding]
        four bytes of data-area checksum
            [treated as a longword for encoding]
        512 bytes of data
            [treated as a block of 512 bytes for encoding]
```

NOTE: The track number and sector number are constant for each particular sector. However, the sector offset byte changes each time we rewrite the track.

The Amiga does a full track read starting at a random position on the track and going for slightly more than a full track read to assure that all data gets into the buffer. The data buffer is examined to determine where the first sector of data begins as compared to the start of the buffer. The track data is block moved to the beginning of the buffer so as to align some sector with the first location in the buffer.

Because we start reading at a random spot, the read data may be divided into three chunks: a series of sectors, the track gap, and another series of sectors. The sector offset value tells the disk software how many more sectors remain before the gap. From this the software can figure out the buffer memory location of the last byte of legal data in the buffer. It can then search past the gap for the next sync byte and, having found it, can block move the rest of the disk data so that all 11 sectors of data are contiguous.

```
Example:

    The first-ever write of the track from a buffer looks like this:

    <GAP> |sector0|sector1|sector2|......|sector10|

    sector offset values:

              11      10       9     .....     1

    (If I find this one at the start of my read buffer, then I know
     there are this many more sectors with no intervening gaps before
```

```
 I hit a gap).  Here is a sample read of this track:

<junk>|sector9|sector10|<gap>|sector0|...|sector8|<junk>

value of 'sectors till end of write':

         2       1     ....    11    ...    3

result of track re-aligning:

<GAP>|sector9|sector10|sector0|...|sector8|

new sectors till end of write:

        11       10       9    ...     1

so that when the track is rewritten, the sector offsets
are adjusted to match the way the data was written.
```

Sector Label Area This is operating system dependent data and relates to how AmigaDOS assigns sectors to files. Reserved for future use.

MFM TRACK ENCODING

When data is MFM encoded, the encoding is performed on the basis of a data block-size. In the sector encoding described above, there are bytes individually encoded; three segments of 4 bytes of data each, treated as longwords; one segment of 16 bytes treated as a block; two segments of longwords for the header and data checksums; and the data area of 512 bytes treated as a block.

When the data is encoded, the odd bits are encoded first, then the even bits of the block.

The procedure is: Make a block of bytes formed from all odd bits of the block, encode as MFM. Make a block of bytes formed from all even bits of the block, encode as MFM. Even bits are shifted left one bit position before being encoded.

The raw MFM data that must be presented to the disk controller will be twice as large as the unencoded data. The relationship is:

1 -> 01	
0 -> 10	;if following a 0
0 -> 00	;if following a 1

With clever manipulation, the blitter can be used to encode and decode the MFM.

Amiga Programming

>$28.95 USA
>$37.95 CANADA

THIRD EDITION

AMIGA TECHNICAL REFERENCE SERIES

AMIGA® ROM Kernel Reference Manual

DEVICES

The Amiga computers are exciting high-performance microcomputers with superb graphics, sound, multiwindow and multitasking capabilities. Their technologically advanced hardware is designed around the Motorola 68000 microprocessor family and sophisticated custom chips. The Amiga's unique system software provides programmers with unparalleled power, flexibility, and convenience in designing and creating programs.

Written by the technical experts at Commodore-Amiga, Inc., who design the Amiga hardware and system software, the ***Amiga® ROM Kernel Reference Manual: Devices*** presents tutorials and detailed examples showing how to use the Amiga's system device interfaces. This new edition has been completely revised and updated for Release 2, the latest version of the Amiga's operating system. It includes:

- A comprehensive introductory section for the novice device programmer
- Complete coverage of all the Amiga's system devices with new information on the enhanced Clipboard, Console, Keyboard, Timer, and Trackdisk devices
- Expanded coverage of Amiga Resources and a new section on the SCSI device
- A complete listing of the IFF (Interchange File Format) specification

For the serious programmer who wants to take full advantage of the Amiga's impressive features, the ***Amiga ROM Kernel Reference Manual: Devices*** is an indispensable source of information on how to use the advanced I/O capabilities of the whole family of Amiga computers.

Cover design by Hannus Design Associates

Addison-Wesley Publishing Company, Inc.

52895>
9 780201 567755
ISBN 0-201-56775-X
56775