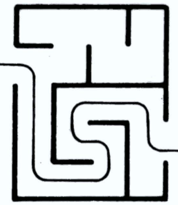
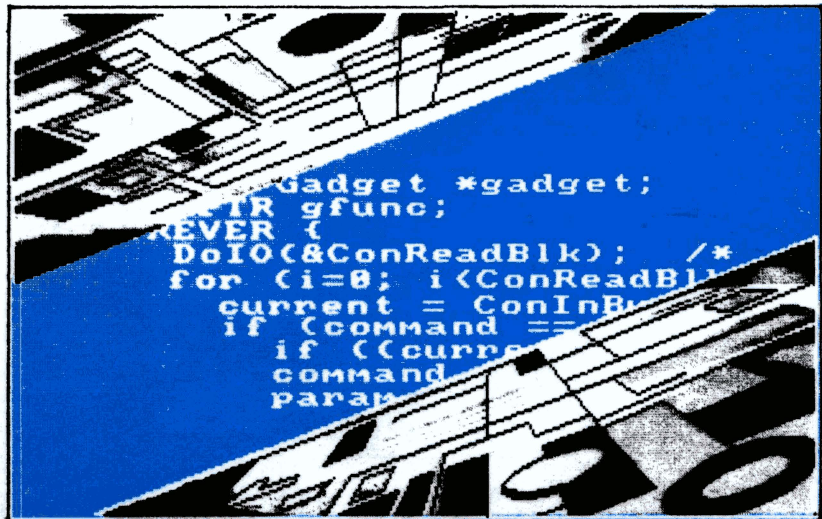
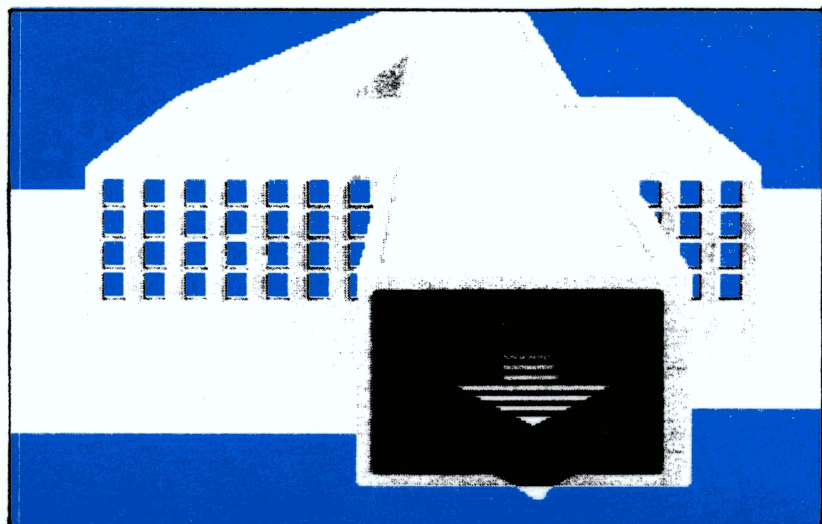
ARIADNE SOFTWARE LTD



The 'Kickstart'™ Guide to the AMIGA™





THE KICKSTART GUIDE TO AMIGA

ARIADNE SOFTWARE LTD

Published by Ariadne Software Ltd,
273 Kensal Road, London W10 5DB, ENGLAND.
Tel 01-960 0203

1st edition full of spelling mistakes - June 1987.
2nd edition with some corrections - September 1987.
3rd edition with various updates - November 1987.

(c) Ariadne Software Ltd 1987.

ISBN 0 9512921 0 2

PRINTED IN ENGLAND
BY

THE DACOSTA PRINT & FINISHING COMPANY
111 Salusbury Road London NW6 6RG
01-969 1111

Text mostly by Dave Parkinson; additional text, example programs and research by Mike Bolley.

Illustrations by Tris Murray (Exec), Hugh Riley (Libraries Architecture), Hanafi Houbart (Into Amiga Devices), Paula Dawson (Keys to AmigaDOS), Phill Legard (Guru Alert), and Shelley O'Neil (Graphics).

Cover design and general inspiration by Chris West.

Thanks to Chris Wood and Gossamer Graphics for help with illustrations and cover art-work, and to Tim Newport for help with printing.

Thanks to Jane Firbank (mighty Compunet editor) for proof reading - hello to all on Cnet!

A lot of other people helped with Kickstart journal. In roughly chronological order, special thanks are due to to Richard Leman (his idea in the first place), Gail Wellington (Commodore originator, regular contributor and strong support), Bill Donald (original editor, regular contributor and number one asker of difficult questions in public houses), Harry Broomhall (official disk wizard), Chris Maciejewski and Tim Bunce (Intuition experts), John Simon Phillips (technical help from Commodore), Dr Tim King (THE expert on AmigaDOS), Mike Todd (champion Fish filleter), Barry Walsh (graphics superstar), and Mark Power (reprographic consultant). Thanks also to everyone who bothered to send us feedback forms, or wrote to us.

KICKSTART is a registered trademark of Commodore Amiga Inc. Amiga is a registered trademark of Commodore Amiga Inc and Commodore Electronics Ltd.

All rights to information published in this book are held by Ariadne Software or by Commodore Electronics. No information may be reproduced or distributed in hard copy or electronically without prior written permission of the copyright holder.

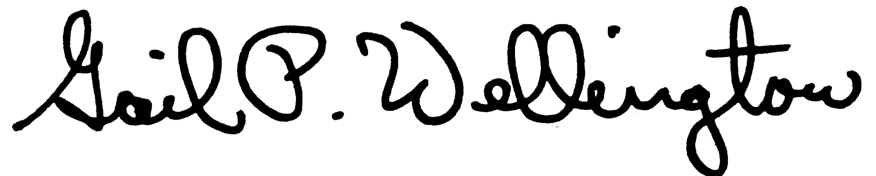
Commodore Electronics Ltd and Ariadne Software Ltd cannot be held liable for any errors in this publication.

A Note from Gail

The Amiga was launched in July 1985 amidst much acclaim. Its first major appearance in Europe was at the European Amiga Developers Conference held at Eastbourne in December 1985. Commodore knew that software was the key to the machine's ultimate success, and the 300 developers who attended the conference had the benefit of talking first hand to the designers of the Amiga.

The "Kickstart" journal was developed to make this sort of information accessible to more people. Commodore are grateful to Ariadne's professionalism in developing the journal for us. Interest continues high in the Amiga family - because of the value of the information in the journals, we are pleased that Ariadne have chosen to publish this book, and we thank them for their continued support of the Amiga.

Whether you program for fun or for profit, this book will improve your understanding of the Amiga.



Gail Wellington,
Director Product & Market Development Group,
Commodore International Ltd.

About The Kickstart Guide to Amiga

It was back in December 1985 that Dave Parkinson and Mike Bolley of Ariadne Software looked at the new Amiga, looked at the huge pile of technical documentation next to it, and thought "'Strewth...".

Before this, Ariadne Software had been involved in assembly code programming on 8-bit micros, such as Commodore PETs (MTC PILOT, MSC PILOT and PETNET), the BBC Micro (NPL's Microtext authoring system and Robocom's Bitstik CAD package), and the Commodore 64 (Microtext again, plus the Compunet terminal). We had also done some 'C' programming on PCs, and hadn't been much impressed.

Our original interest in Amiga was due to an involvement in authoring systems for interactive video and training, and a desire to go a lot further than had been possible on the BBC Micro and Commodore 64. Some early experiments suggested that PCs did not offer sufficient power and flexibility to do this; the Amiga looked a lot more promising. With this in mind, Dave Parkinson went off to the Amiga Developers Conference in Eastbourne, and later the decision was made to purchase an Amiga - Mike Bolley must be one of the few people to buy an Amiga without ever actually having SEEN one, his decision being based solely on the technical information contained in the Eastbourne course notes!

Having bought the machine, we found ourselves at the start of a very considerable learning process. 8-bit machines and PCs we understood very well, but this was something else - what when they were at home were "pre-emptive scheduling" or "round robin time slicing"? Finding answers to questions like these involved us in an extended process of reading, experimentation, discussion with other Amiga developers, and buying drinks for people who knew mainframe and mini operating systems - "Alright Hugh, come clean, what exactly IS a "lock" precisely?".

It was Richard Leman of JCL Software who first suggested that we should use the learning experiences of ourselves and other developers as the basis of a European Amiga "technical journal". We took this idea to Gail Wellington of Commodore Electronics, who responded enthusiastically, and before long had obtained company support for the project. With Gail pushing, Commodore supplied us with backing in the form of a bit of money, a lot of technical support, and considerable help with distribution, the journal going free of charge to all registered European developers. To help us get started, Bill Donald came in as our original "general editor"; also helping out at this stage was artist/programmer Hanafi Houbart, who - amongst other claims to fame - thought of the title "Kickstart".

In all, we produced six issues of Kickstart journal before we got too busy with other things - notably developing video and authoring software for Amiga! Each issue contained a feature article concentrating on some particular aspect of the machine, plus articles by other people, a series on 'C' on Amiga, and a "Crosstalk" section for information exchange between developers. Reaction to the journal was very favourable, with positive feedback being received from Holland, from Germany, from Ireland, from Monaco, from the UK, from Israel, from Switzerland, from New Zealand, and from France. From further afield, we heard from South Africa, from Australia, from the States and from Canada - the latter asking if we wouldn't mind considering them as an honorary part of Europe!

Now, with the release of the new A2000 and A500 machines, a "second generation" of progammers and developers are approaching the Amiga, often from a similar background to ourselves. With them in mind, we have taken the feature articles and 'C' series from Kickstart, revised and updated them, and added new material - the result is this book. This is NOT intended as a replacement for the official Amiga technical manuals, and it WON'T turn you into a demon Amiga programmer overnight; it DOESN'T have very much to say about Amiga hardware, or higher-level aspects of the system software such as animation or speech. It DOES aim to provide a "step up" to the Amiga from other machines, in the form of an introduction to 'C' programming on Amiga plus a comprehensible account of how the machine works in terms of Exec, AmigaDOS, and graphics - once you understand these, the rest is pretty easy. We hope you find this book useful - enjoy the Amiga!

David Parkinson,
Kickstart Editor,
Ariadne Software Ltd.

Contents

Part 1 - Introducing the Amiga. 1

Appendix - introducing the 68000 13

Part 2 - The Kickstart Guide to Amiga. 23

Section 1 - Exec. How to do several things at once while doing one thing at a time. 23

Section 2 - Libraries. How to call a routine without knowing where it is. 66

Section 3 - Devices. How to perform IO without worrying too much what to. 108

Section 4 - Aspects of AmigaDOS. 146

Section 5 - Serial port debugging, and the Joy of Wack. 169

Section 6 - Introducing Amiga graphics. 192

Part 3 - Getting Started in C. 227

Section 1 - Introducing C. 227

Section 2 - Elements of C. 238

Section 3 - Structures and Pointers. 244

Section 4 - Arrays and Strings. 251

Section 5 - Getting finished in C. 257

Appendix 1.2 libraries summary. 265


```
HowToReadThisBook()
{
     ReadPart(1, CAREFULLY);

     if (YouThinkYouKnowC) {

          ReadPart(2, CAREFULLY);
          ReadPart(3, QUICKLY);

     } else {

          ReadPart(3, QUICKLY);
          ReadPart(2, CAREFULLY);
          ReadPart(3, CAREFULLY);
     }
}
```

PART I - Introducing the Amiga

The Commodore Amiga is an amazing machine in terms of its clever hardware, its multi-tasking software, and its advanced WIMP (Window Icon Mouse & Pointer) user interface. The purpose of this part of the Kickstart Guide is to give a general overview of the machine; we will then go into detail about how various bits of it work in Part 2.

The three areas in which the Amiga excels - the hardware, the multi-tasking, and the user-interface - are in fact very closely related. Clever hardware like the blitter or bimmer (block image manipulator) takes a lot of the burden of maintaining a complex colour display away from the CPU and also handles things like audio output - this enables the CPU to be used for other things, such as running a sophisticated multi-tasking message-passing "operating environment". (In this context it's worth noting that Commodore-Amiga had the blitter in 1984, and it's taken them since then to get the software right - other companies who are just developing blitters now have therefore got a bit of catching up to do!)

The multi-tasking has many applications on Amiga - including nice things like being able to edit one program module while you compile another and perhaps hard-copy a third - but its real significance lies in its application to "Intuition", the Amiga's user-interface. Earlier WIMPs-based machines offered the end-user a nice easy time of it (or not so nice and not so easy in the case of some systems we could mention); however this was achieved at the expense of considerable sweat and grief on the part of the application programmer, who had to worry not only about the actual application, but also about things like "what do I do if the system tells me the user is trying to resize my window?".

The Amiga is the first machine to take this burden away from the application programmer, and therefore to provide a WIMPs user-interface in a sensible civilised manner. On the Amiga, an application program can get on with whatever it is supposed to be doing, while another task worries about keeping the user happy - in fact Intuition's "input-event handler", running as part of a task associated with the "input device" on Amiga. This means that you don't have to worry about things like windows moving and resizing AT ALL - unless you specifically want to. Instead, you just tell Intuition what you DO want to hear about - user selecting a particular menu item, user clicking on a particular "gadget", or whatever. You can then get on with something else, or sit in a comfortable wait-state, waiting for Intuition to send you messages about events of interest to something called your IDCMP (for "Intuition Direct Communication Message Port"). Compared to programming other WIMPs machines, this is absolute bliss - and it means that while Amiga development is still a laborious business compared to cobbling something together on a Commodore 64, it can be done in a fraction of the time it takes to get a similar result on other WIMPs computers!

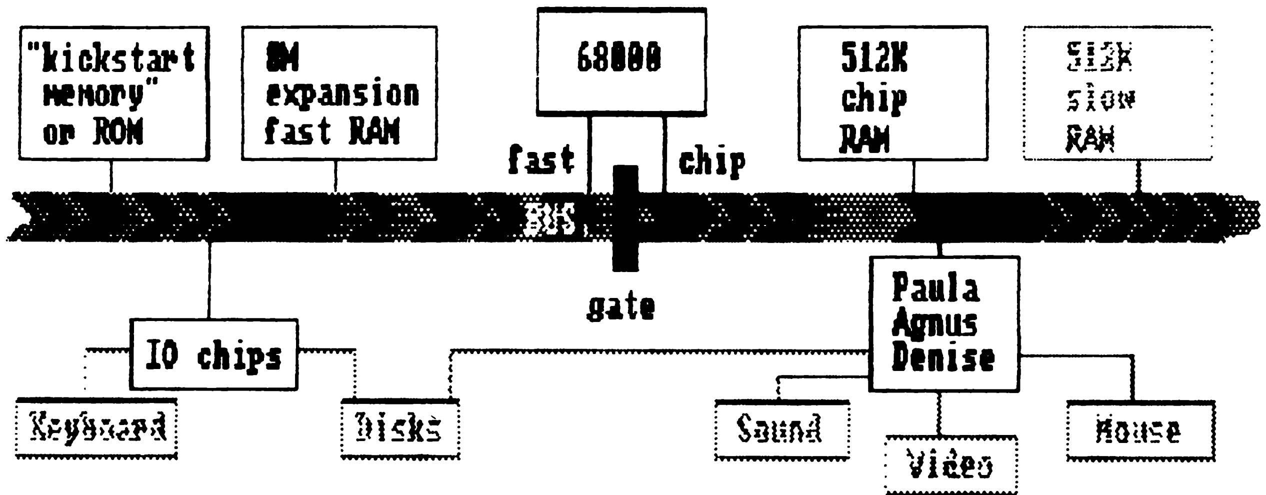


Figure 1 - Amiga Hardware Overview

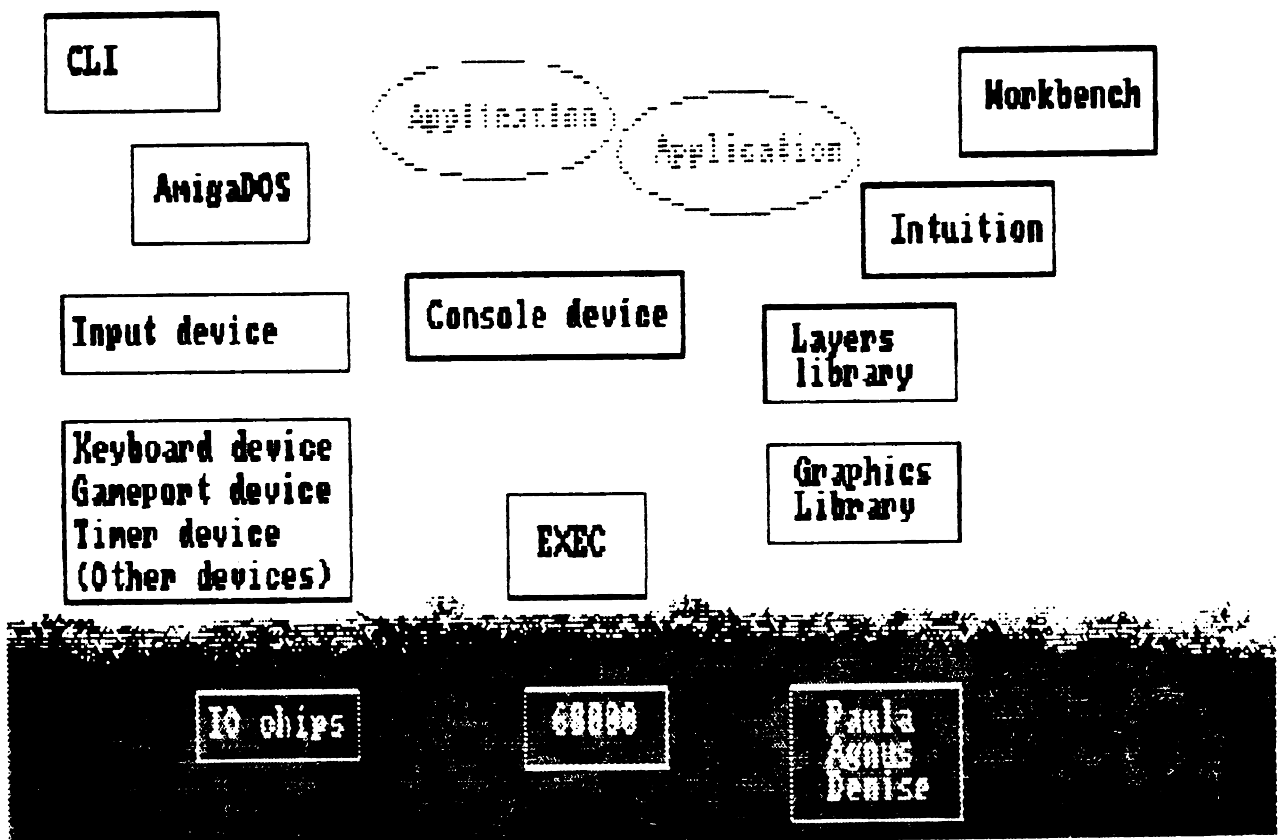


Figure 2 - Amiga Software Overview

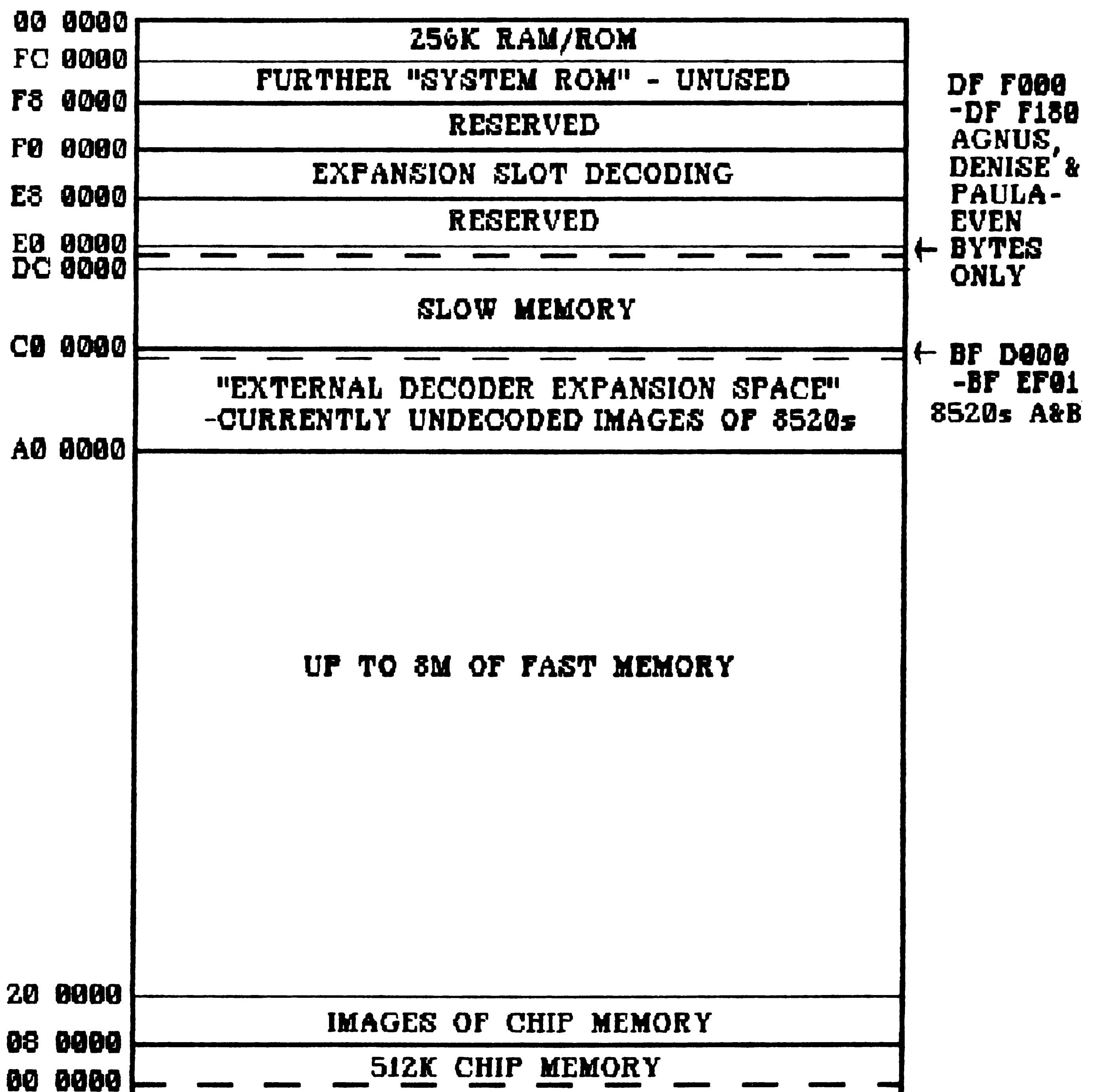


Figure 3 - Amiga Memory Map

Amiga Hardware Overview

A diagram showing an overview of Amiga hardware is given in Figure 1; a memory map is given in Figure 3. Basic hardware elements are as follows:

68000 — Amiga Central Processor Unit, Motorola 68000 running at 7.3 MHz. A later chip than the 8086 series used in PCs, with more sensible memory management, more powerful interrupts, and a more rational instruction set. Can be upgraded to 68010 or 68020, with optional floating point co-processor, for REALLY amazing performance.

PAD — Paula, Agnus and Denise - known collectively as "the PAD" - the "clever chips" responsible for a lot of Amiga special features, especially involving sound and graphics. Capable of running "in parallel" with the CPU - using alternate (even) clock half-cycles when the 68000 isn't accessing external memory - thus taking a lot of processing burden off the 68000 and giving the Amiga much better performance than cruder machines with slightly faster clock rates.

gate — Divides the Amiga system bus into "fast" and "chip" memory. "Chip" memory is the bottom 512K, which is capable of being accessed both by the 68000 and the PAD; "fast" memory is up to 8 megabytes of further RAM which is not accessible to the PAD, and therefore cannot be used for graphics screens, accessed by the blitter, etc. The reason for this is that under some circumstances the PAD "cycle steals" from the 68000 - ie stops it from accessing memory for a while, while the PAD is busy fetching data for a high resolution screen line, or doing a data-move using a "nasty" blitter. ("Nasty mode" is a blitter mode in which it cycle steals a lot, eg when doing block memory transfers - not unreasonable, given that the blitter is a much more efficient data mover than the CPU!) The presence of the gate in the bus means that the PAD can be cycle stealing like mad in the bottom 512K of memory - eg in order to do a complex high res animation - but WITHOUT blocking CPU access to fast memory; the CPU can therefore continue to operate at full speed, until such a time as it needs to access chip memory. A sensibly configured Amiga has AT LEAST as much fast memory as chip memory, and preferably more.

512K RAM — "Chip memory", accessed both by 68000 and the PAD.

512K RAM — Extra 512K of "slow memory" on A500 (optional) and A2000 only; used to move system structures (eg Exec library stuff and the supervisor stack) out of chip memory, freeing more chip memory for use by graphics, sound etc. See Appendix 2.

8M RAM	Up to eight megabytes of "expansion memory" - fast memory, accessed by 68000 only. Has to be external on the A1000 or A500; can be fitted internally on the A2000.
Kickstart memory	Used to store vital Amiga system software such as Exec, graphics and layers libraries, Intuition, and most of AmigaDOS. On A1000, this software was loaded on power-up from a "Kickstart" disk into a special 256K region of additional RAM known as "Kickstart memory" or "RAM/ROM" which was then write-protected - this was to enable Commodore-Amiga conveniently to issue software updates. On the A2000 and A500 this is no longer necessary, and the original RAM/ROM has been replaced by ordinary ROM. Note that this is not the whole story - to actually operate the Amiga, more software is needed such as the rest of DOS and the "Workbench"; this is loaded into ordinary RAM from another disk called "Workbench".
IO Chips	Two 8520s, similar to the CIAs (Complex Interface Adaptors otherwise known as Completely Incomprehensible Adaptors) used in the Commodore 64. Contain two IO ports each plus various clocks and timers, all of which are "used up" by bits of system software such as the timer device. Handle serial port, parallel port, keyboard, and disk control - though disk DMA is looked after by the PAD.

More about the 68000

It is not our intention to consider 68000 programming in much detail in this guide; the interested reader will find plenty of books on the subject. However, it's useful to know a bit about it, even if only to gain insight into what your "C" compiler is up to - for this reason, a quick overview of the 68000 is provided as appendix 1 to this introductory section.

More about the PAD

As mentioned above, Paula Agnus and Denise are responsible for a lot of the high performance of the Amiga. Functions looked after by the PAD are roughly as follows:

1. CPU control. The PAD looks after the 68000 on Amiga, by generating its DTACK signal usually provided by external hardware to indicate a successful data transfer, by "blocking" its access to external memory when it wants to cycle-steal, and by controlling its interrupts. Interrupts on the Amiga are looked after by Paula - there are sixteen possible interrupt sources, which are two external hardware, one vertical blank, one copper (video beam reached a

specified position), four audio channels (audio block done), one blitter (blitter finished), two disks (sync found, disk block finished), two CIA for keyboard and timers, and two serial port (receive buffer full, transmit buffer empty). Paula looks after watching and prioritising these interrupt sources, and deciding whether to interrupt the CPU, and if so with what priority.

2. DMA control. There are twenty five "dedicated" Direct Memory Access channels on Amiga, used for direct access to chip memory by the PAD without involving the CPU. ("Dedicated" means these channels are tied to a particular purpose - eg "audio channel one" - and that you can't swipe them for use for something different!) DMA is used on Amiga for bitplane access (ie the screen - six channels), for sprite data access (eight channels), for copper instruction-fetch (one channel), for the blitter (four channels), for disk DMA (two channels), and for audio (four channels).

3. Playfield and sprites. The PAD handles forming a basic "playfield" (ie screen display) by fetching data from a number of "bitplanes" in chip memory, and interpreting it using internal colour registers - this is known as "colour indirection". The PAD can also handle up to eight hardware sprites on top of the basic playfield, which can be up to sixteen pixels wide and any number of columns deep; sprites can be "joined together" for greater width or more colours, and the apparent number of them can be increased considerably by clever tricks using the copper. An obvious example of sprites on the Amiga is the Intuition pointer.

4. Copper - beam-synchronised graphics co-processor. You can think of this as "watching" the video beam go down the screen, while following a simple program known as a "copper list" telling it what to do when the beam reaches specified positions - "wait till the beam reaches so-and-so then do this". The copper is capable of changing internal PAD registers directly - eg changing playfield pointers for split screen, or sprite pointers for sprite multiplexing - or of affecting external memory by issuing a CPU interrupt, or using the blitter.

5. Blitter or Bimmer. Block Image Manipulator with three input channels and one output. Capabilities are as follows:

   a. Can move data around VERY fast in chip memory - can read/write a 16-bit chunk every 280 nano seconds, though in practice this is slowed down by competition for DMA with the rest of the PAD.

   b. Can perform LOGICAL OPERATIONS - two of its inputs have "barrel shifters" on them to shift data left or right, and the three input streams can be combined in any one of 256 possible logical operations, expressed by

blitter "minterms". Input might consist of a graphics object, a mask, and screen background; output might consist of the current screen bitplanes - this is the basic technique used for animation with "blitter objects" or "Bobs".

c. Can perform LINE DRAWING or AREA FILL operations directly into chip memory. This is also blindingly fast - the system software using the blitter can draw up to about four thousand lines a second!

6. Audio channels. The PAD has four audio channels, each of which looks at a bit of external memory, interprets the contents as a digitised waveform, and outputs the result as audio. This "digitised waveform" approach is very powerful, and is responsible for the Amiga's remarkable sound and speech capabilities.

7. Disk. Transfer to/from disk is a whole track at a time, using DMA; the system doesn't even wait for disk sync, but instead just reads in the data wherever from wherever the disk head happens to be, then sorts it out sensibly using the blitter. This leads to very fast disk access - a good thing, given the high overhead involved in the way AmigaDOS handles directories!

Amiga software overview

A diagram giving a rough overview of Amiga system software is given in Figure 2. It can be seen that this isn't much like a conventional "Operating System"; instead we talk about an Amiga "Operating Environment" consisting of a large number of intercommunicating elements, organised as "libraries", "devices" and "resources" - more about these below.

There have in fact been three major releases of Amiga operating software, which can be roughly categorised as follows:

V1.0 mid 1985	The best that we could do given the timescale.
V1.1 early 1986	The best that we could do, with most of the bugs taken out.
V1.2 early 1987	What we should have done in the first place, only we needed to do versions 1.0 and 1.1 to find out.

Version 1.2 (Kickstart 33.180) is now occupies 256K of ROM in the A500 and A2000.

Tasks and processes

The basic unit of multi-tasking on the Amiga is a "task" - this can be thought of as a 68000 program which is being fooled that it has a whole machine to itself; different tasks are actually swapped in and out by Exec running "on the interrupts" in supervisor mode, as explained in detail in Part II Section 1. Tasks usually spend most of their time in "wait states" - ie fast asleep until something of interest happens; this something of interest is usually the arrival of a "message" from another task, asking it to do something. Messages are sent to tasks' "message ports" - eg you can send a message to the "console device" asking it to output some data. This is used to implement asynchronous IO amongst other things - you can send another task a message asking it to do something, then get on with something else until the task indicates it has finished, which it does by "replying" your message.

Tasks are used on Amiga for system use, and also to run application programs; a task in this context is part of a higher level AmigaDOS concept known as a "process", which consists of a task plus a lot of other stuff, to do with default IO channels etc.

Libraries

Amiga system software is organised into "libraries" - these are essentially a load of routines starting with a jump table. These routines can be called from other libraries, or directly from application programs; calling library routines is the normal way of getting things done on Amiga, sending messages being reserved for special purposes such as asynchronous IO. A full account of libraries is given in Part II Section 2, and a summary of all routines available in 1.2 system libraries is given as an appendix to this book. Some key libraries are as follows:

Exec The "multi-tasking executive" written by Carl Sassenrath. In charge of 68000 interrupts; the lowest level of Amiga system software, which looks after everything else.

Graphics Amiga graphics routines by Dale Luck; in charge of the PAD graphics capabilities, including the blitter. Contains a full set of routines for screen management, plus drawing routines for points, lines, area-fills, flood fills, circles and ellipses. Also contains routines for text - text is a special case of graphics on Amiga!

Layers Also by Dale Luck; work in very close conjunction with the graphics libraries - routines which allow a single drawing area to be treated as a number of overlapping layers, such as Intuition windows.

Intuition Designed and originally coded by R.J. Mical; revised for V1.2 by Jim Macraz. Routines which handle the user-

interface, in the form of screens, windows, menus, gadgets, requestors etc. Appears both as a library, and as an "input handler" connected to the "input device" task - in its latter form, is capable of handling things like window moving and resizing, menu selection etc, without involving the application program. Generally speaking, you ask Intuition to do things by calling routines from the Intuition library; it tells you things of interest (eg "The user has selected this gadget") by sending messages to something called your IDCMP - for Intuition Direct Communication Message Port. (Intuition will generally create an IDCMP for you and a "reply port" for its own use when you open a window, so there's no need to worry about this too much.) Intuition is used heavily by another important piece of Amiga software called the Workbench - this is an AmigaDOS process which uses Intuition to provide the user with a standard way of performing disk and file operations, and of starting application programs.

AmigaDOS An Amiga "late entry", brought in when an original DOS project collapsed - written by Dr Tim King and others of UK software house Metacomco. Based on the original Tripos operating system developed in Cambridge in the late seventies, designed to be as small and portable as possible, at the expense of "luxury" features found in larger systems like Unix. Handles files, devices and processes, including launching application programs. Can be called by other processes such as Workbench; alternatively can talk to you directly using a special form of process called a "CLI" (for Command Language Interface). A bit of an odd man out - we once described it as fitting in with the rest of the system like a man in a dinner suit at a beach party, but subsequently relented towards it. Written in BCPL - a language unknown to Americans which is a forerunner of C.

A lot of rubbish has been written about AmigaDOS. Some sources overrate the DOS by trying to credit it with everything the Amiga can do - in fact AmigaDOS only accounts for about 40K of the complete Amiga 256K ROM space. Other sources go to the opposite extreme of blaming everything they DON'T like about the Amiga on the DOS and on BCPL, which is pretty silly. The original Amiga "DOS" project was intended to produce just a "filing system and process manager" integrated with the rest of the environment; AmigaDOS does this quite successfully, while adding its own "devices" (CON:, RAW: etc) and the CLI environment as a bonus. The former are of dubious benefit, and we would argue that you can write better Amiga programs by ignoring them and going directly to other parts of the environment (eg "console.device") as originally intended - unfortunately the "standard functions" in things like Lattice C don't do this! On the other hand, the CLI is DEFINITELY a virtue - while it is easy enough to criticise the CLI, try and imagine Amiga development without it.

There are many other Amiga libraries; for information on these, see Section II part 2, and the Appendix.

Resources and devices

Resources and devices are two special sorts of Amiga software entity, both based on the fundamental structure of a library. A resource is a rather low-level object, concerned with "contention management" - the function of a resource is to grant or forbid access to a particular bit of hardware, depending on what the rest of the system is up to. Resources are generally looked after by other bits of Amiga system software; you only have to worry about them yourself if you want to directly access a bit of hardware such as the parallel port, in which case you should first "open" the corresponding resource to avoid contention problems - "misc.resource" in this case. A device is a special sort of library concerned with IO on Amiga; many devices also have tasks associated with them, so that they can operate asynchronously of the calling program if necessary. A full account of Amiga devices will be found in Part II section 3; a summary of some important ones is as follows:

Trackdisk device Low-level disk IO; used by AmigaDOS.

Keyboard device Low-level keyboard input; works in terms of "raw" keyboard events such as key-pressed/key-released.

Gameport device Low-level mouse input.

Timer device Low-level timing - uses the 8520 timers.

Input device A very important one this. A "cunning" device with an associated task; handles coordinating input data from keyboard device, gameport device and timer device, and passing it on to a chain of "input-handlers" - notably the Intuition input handler, and/or the console device.

Console device A "high level" device - takes input from the input device and performs output to a specified window using the graphics library text primitives, in order to give a "virtual terminal" capability. A full ASNI standard terminal with a whole range of controls and escape sequences; used by AmigaDOS "RAW:", or can be accessed directly. Note that, contrary to a once widely-held belief, the console device and hence AmigaDOS RAW: does NOT return "raw" keycodes unless you explicity ask it to - instead you get nice ASCII values, and "escape" sequences headed by a CSI (Command Sequence Introducer) character. Information which can be passed back from the console device using CSIs includes reports of mouse-movement, gadget selection etc; thus talking to the console often provides an alternative to using an Intuition IDCMP.

More information on these devices, including a full discussion of the different ways they can be connected together for different ways of doing IO on Amiga, will be found in Part II section 3. Other devices include audio, narrator, serial, parallel and printer - for information on these, see the ROM kernel manuals.

Development on Amiga

An account of development on Amiga using the C language is given in Part III - "Getting Started in C". A summary of important development tools is as follows.

Compilers Used to convert source-code to "object module" format. Favourite development language is C, but many others are available, including Pascal, Modula 2, LISP and APL, to name but a few. Original "official" Amiga C compiler was Lattice V3.03, which was followed by Lattice V3.1 and now by Lattice 4; an alternative (with many adherents) is Manx Aztec C.

Assemblers Also used to convert to object module format; this format is the same for assembler and compiler output, so it's quite easy to mix the two, eg in order to re-code time-critical routines in assembler. A C function call Fred() results in a subroutine call JSR _Fred; _Fred can be written in assembler, and joined together with the C calling function using the linker. Parameters can be passed from C to assembler by reading them off the stack - for more information on this, see "Getting Started in C" later in this publication. Original "official" assembler was Metacomco ASSEM; while this is still the standard assembler, some good alternatives are now available, including some in the public domain. ASM68K on the Fish disks is worth investigating.

Linkers Used to join together object modules to form load modules, together with standard startup-code, and further routines for standard functions from linker "scanned libraries". Original Amiga linker was Alink (versions 1.0 and 1.1); an alternative which started on the public domain and is now used as standard by Lattice, is Blink from the Software Distillery. Original (Lattice V3.03) startup modules were AStartup.obj or LStartup.obj; new (Lattice V3.1) startup module is c.o.

Monitors Original Amiga monitor was Wack, written in C by Carl Sassenrath, to run on the original Amiga development machine, which was a Sun workstation. This was then ported across to the Amiga, to form a cut-down version re-coded in assembler and included in the ROM called ROMWack, and a full version known as GrandWack - this was released undocumented with version 1.0, and is now known by us as OldWack. Metacomco were then given a contract job to clean up Wack to give Wack 1.3, known to us as NewWack - this is supposed to be included on the 1.2 developers toolkit disk, which is still (Nov '87) being eagerly awaited. Meanwhile, other monitors have been developed as commercial packages - Lattice now bundle one called Metascope, which makes very good use of Intuition to provide multiple dynamic windows into memory to aid you in debugging. Metascope is limited however - at least in the last version we saw - in that it tends to fall over when asked to look at something like a sub-task, a library or a device - a new version of Wack (capable of coping with the new hunk-types introduced by Lattice) is therefore badly needed!

Other tools There are many, many other development tools available on the Amiga. These include alternative "shells" - such as the Dillon shell or the Metacomco shell - which give a much more civilised alternative to the standard Amiga CLI. Other tools ease the task of designing things like menus then generate the corresponding C source-code, or take "brushes" from Paint packages and generate the corresponding "Image" structures for use in things like Intuition gadgets. Some of these are commercial products - eg the Metacomco Shell or Power Windows packages - while many others are public domain or shareware, and can be found on the Fish disks.

Documentation Originally, Amiga documentation was published in several large volumes and circulated by Commodore; 1.1 versions of this were then given to commercial publishers Bantam (AmigaDOS) and Addison Wesley (all the others). If you are going to do serious Amiga development, then the following documentation is essential:

ROM kernel manual Vols 1 and 2	(Addison Wesley)
Intuition manual	(Addison Wesley)
AmigaDOS User Guide & ref manual	(Bantam)

Optional but useful is the Hardware manual, also published by Addison Wesley. Absolutely crucial are the updates to these manuals provided in the 1.2 enhancer documentation from Commodore; also CRUCIAL are the disks containing 1.2 commented h-files and full library and device routine descriptions ("autodocs"). We understand that the 1.2 enhancer manual is currently being shipped with Amiga 2000s, while the autodocs are available in the States as the "Native Developer Update" which costs $20 from Commodore Amiga Technical Support (CATS), 1200 Wilson Drive, Westchester, Pensylvania 19380. (Availability in other countries is unknown - ask Commodore.)

Examples etc Best source of these is the Fish disks - over eighty disks of public domain or shareware material collected from US bulletin boards etc by Fred Fish. Original Fish disks - or selections known as "Filleted Fish" - can be obtained from most Amiga user groups, and from bulletin boards. (UK readers should contact ICPUG or AUG, or try Ariadne Software.) Fish disk material can be divided into utilities and examples; amoungst the former, we would particularly recommend the gi brush-to-image converter from Fish 13 (though it seems to work with DPaint 1 only), and the ASDG shareware recoverable RAM disk from Fish 58. The latter allows developers with plenty of memory to load compilers, h-files etc into a special form of RAM disk which can (usually) be resurrected following a "Guru meditation" system crash, which is a great time saver! Fish disk examples include plenty of graphics stuff, and quite a bit on DOS and Intuition. We would advise you to look at things like the nice bi-scrolling disk directory from Fish 35 - believe us that if you decide to write all this sort of thing yourself from scratch, it is going to take you AGES!

Appendix 1 - Introducing the 68000

The objective of this appendix is to give a very brief overview of the 68000 - for more information see any of the many books now available on the subject. Some tables summarising 68000 registers, addressing modes and op-codes are provided at the end of this appendix; further discussion of various aspects of the chip will be found in Part 2, where they are discussed in relation to various software aspects of the Amiga.

68000 registers

A diagram showing 68000 registers will be found in Table 1. Registers are as follows:

D0-D7 Eight general purpose 32-bit data-registers. Can be addressed as byte, word, or long-word - eg MOVE.L #0,D0 will zero the whole of D0, while MOVE.B #0,D0 will zero only bits 0 to 7. Generally used as accumulators or index registers.

A0-A6 Seven general purpose 32-bit address-registers. Can be addressed as word or long-word; generally used as pointers or index registers. Behave very similarly to the data registers a lot of the time, but with some subtle differences - eg 16-bit quantities tend to get "sign extended" when loaded into address registers, and operations on address registers tend NOT to affect the processor status flags, so watch it.

A7 Address register seven = stack pointer, for normal downward-growing stack. In fact the 68000 has **two** stack pointers, for use in "user mode" (USP) and "supervisor mode" (SSP) respectively - roughly speaking, user mode corresponds to normal operation, while supervisor mode is a special state entered when servicing interrupts. The right stack pointer appears automatically in A7 when the 68000 swaps from user mode to supervisor mode, or vice versa.

PC Twenty-four bit program counter, to address up to 16 megabytes of memory. Only (only!) eight and a half to nine megabytes are easily accessible for RAM on Amiga, though you could probably get hold of more than this with a bit of hardware effort.

SR 16-bit status register, divided into user-byte and system-byte; bits in the system byte can only be altered when the 68000 is in supervisor mode. User flags are Carry C, Overflow V, Zero Z, Negative N, and Extend X, the last being similar to the carry-flag but not affected by as many operations, which is handy for multi-precision arithmetic. System flags are a three-bit "interrupt mask" indicating what "level" of

interrupt the 68000 will respond to, plus a supervisor-mode flag S, and a trace-mode flag T - the latter is used in a special mode of operation where a trap (software interrupt) is forced after every instruction, which comes in handy for debugging by single stepping.

68000 address modes

The 68000 supports eleven different address modes, as shown in table 2. The instruction set is quite "orthogonal" meaning that generally speaking you can use any address mode with any instruction, assuming it makes sense to do so. Thus there is no difficulty for example in performing

```
JSR -96(A6)
```

meaning call a subroutine 96 bytes below the "base address" currently in A6 - this is in fact used a great deal in the Amiga, as we shall see in Part II section 2 on "libraries". Note that there is nothing corresponding to 6502 page zero on the 68000, so if you want to use something as an address pointer you generally have to get it into an address register. (Page one isn't special either - the 68000 has a 32-bit stack pointer, meaning the supervisor or user-stacks can be of any size, and located anywhere in memory.)

References to external memory, like references to data-registers, generally come in byte, word, and long-word (32-bit) varieties. Note however that the way that memory management is handled on the 68000 means that the chip is NOT happy performing word or long-word access on an odd-byte boundary - an attempt to do so results in a 68000 "trap" which on the Amiga results in a crash with "guru meditation" number 3.

68000 interrupt handling

The 68000 boasts very powerful interrupt handling, which is known to Motorola as "exception processing" - presumably just to be different. There are three interrupt lines, providing levels of interrupt from zero (no interrupt) to seven (non-maskable interrupt). When servicing an interrupt the 68000 generally sets its "interrupt disable" mask in the system part of its status register to the same as the level of the interrupt being processed; this means that a level five interrupt can be interrupted by levels six or seven, but not by one to five.

Besides normal hardware interrupts, exception processing can also be caused by other "external" events in the form of bus errors or reset, or by a whole variety of "internal" events such as addressing errors, privilege violations, illegal or unimplemented op-codes, divide by zero, or by one of sixteen special TRAP instructions which force exception processing, in a way a bit similar to 6502 BRK.

An exception on the 68000 causes the CPU to push status register and program counter to its current "user stack", to enter supervisor mode, then to jump by way of an appropriate vector in the bottom 1K of memory. The 68000 has some very fancy ways of handling interrupts indeed, but these are not used on Amiga, which handles trickery on the interrupts outside the CPU in the PAD as explained above. As used in the Amiga, the different levels of hardware interrupt, and the various software interrupts and traps, simply correspond to different vectors in the bottom 1K of memory, which point to different entry points in Exec. The fact that the "fancier" interrupt modes aren't used in Amiga means that the top part of the bottom 1K isn't doing anything; this is reserved for use as work-space by the Amiga monitor, Wack. Note that the vectors in the bottom 1K include initial SSP and initial PC values for use during reset; things are got going in a sensible manner on Amiga by having "boot" ROM switched into this area during reset - this ROM also handles loading "Kickstart" into RAM/ROM on the A1000.

We shall consider exception processing on the 68000 in more detail in Part II section 1, when we look at multi-tasking and Exec. For the moment, we shall content ourselves by mentioning a difference in character between 6502 interrupts and exception processing on 68000. On the 6502, interrupt processing can generally be thought of as a "slave" to the main processing, that wakes up every now and again and worries about boring "background" matters like whether the user is typing on the keyboard. On the 68000, exception processing can be thought of as "master" rather than "slave"; an exception puts the 68000 in a special "supervisor" mode, with its own private "supervisor stack", which is independent of anything else going on in the machine; this means that it can take a look at what's going on and mess about with it if it feels like it, such as stopping one program ("task") running and starting another - more about this later when we discuss multi-tasking.

68000 instruction set

A summary of the 68000 instruction set can be found in Table 3. Note that there are only 56 basic instructions, which makes life reasonably easy. However, many of these come in various different "flavours" such as byte, word, long-word and quick:

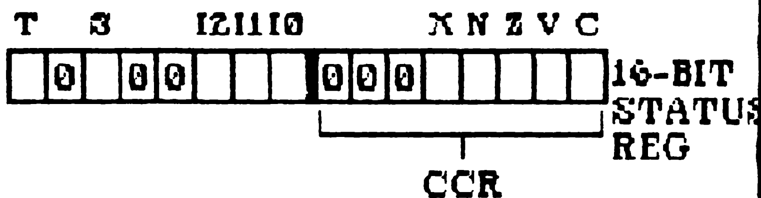
```
MOVE.B    #0,D0    zero bits 0 to 7 of D0
MOVE.W    #0,D0    zero bits 0 to 15 of D0
MOVE.L    #0,D0    zero bits 0 to 31 of D0
MOVEQ     #0,D0    quick move byte with sign extension - in
                   this case will zero all of D0. Occupies
                   only one instruction word.
```

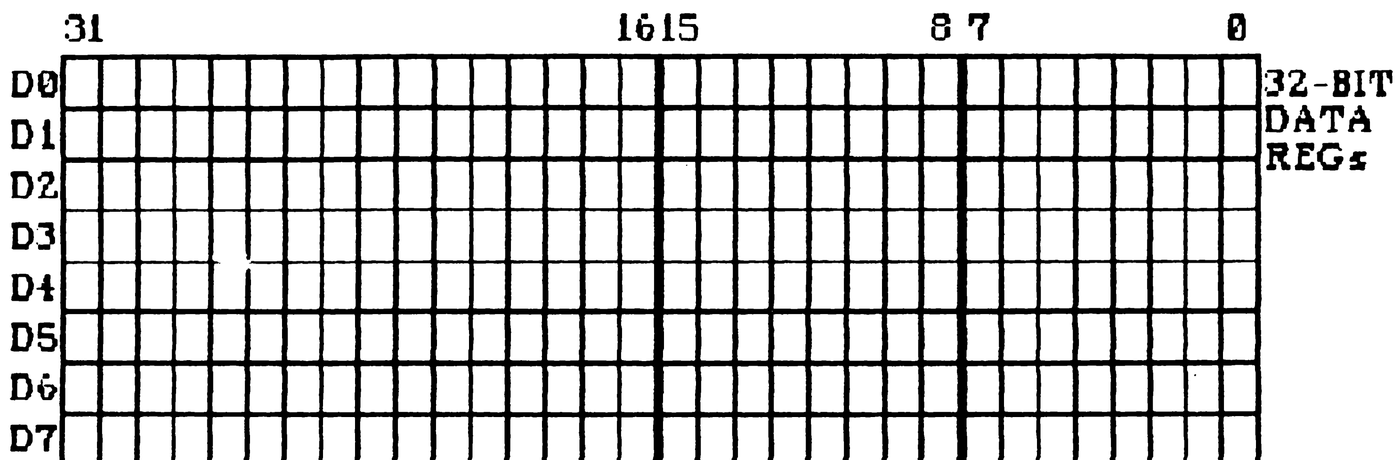
Most of the instructions are fairly self-explanatory. MOVE is probably the most commonly encountered instruction; the use of different address modes with MOVE allows data to be moved between

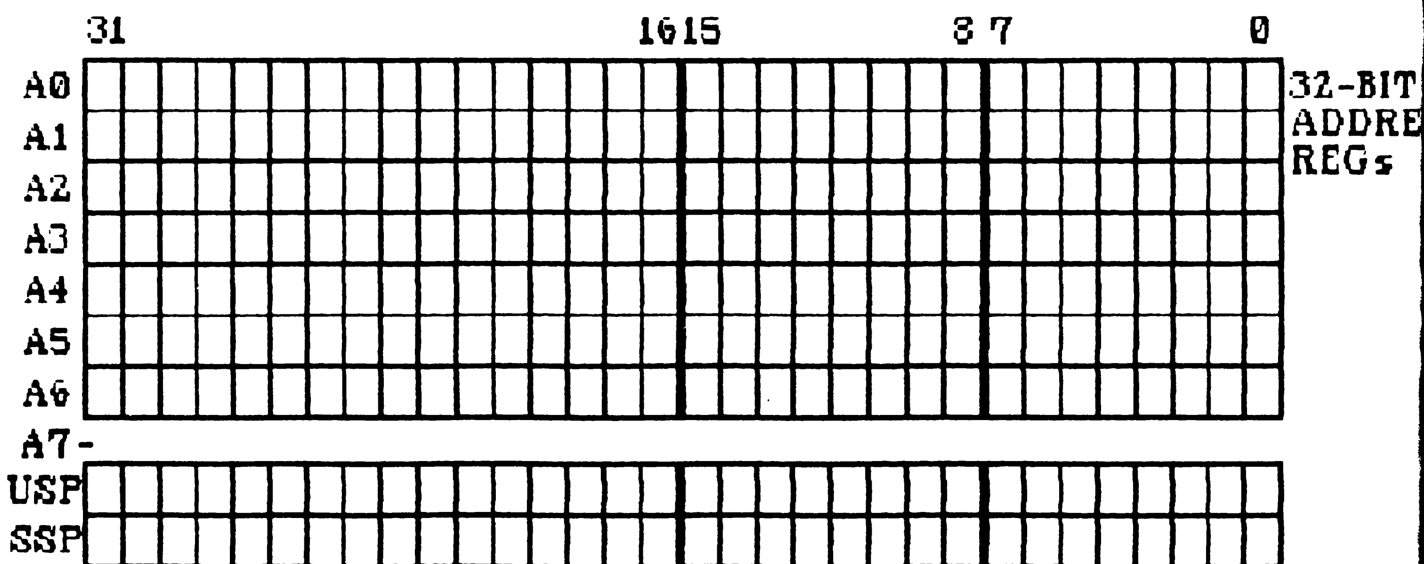
registers, between registers and memory, and directly from memory to memory, with various indirection and indexing options. MOVEM - for move multiple - allows various registers to be specified together in a single instruction, eg for the purpose of pushing them to the stack:

```
     MOVEM.L D2-D7/A6,-(A7)    ; push data regs D2 to D7 and
                               ; address reg A6 to the stack.
```

Other facilities worth noting are signed and unsigned multiply and divide instructions, a variety of branch instructions (eg BSR) which together with "PC-relative" address mode make it fairly easy to write relocatable code (unnecessary on Amiga since any position-dependence can be fixed up by the AmigaDOS loader), and a variety of "test and set" instructions - these allow you to do things like checking if a flag bit is set already and set it if not in a single "atomic" (uninterruptable) operation, which comes in handy in a multi-tasking system. Finally, note the LINK and UNLK instructions - these allow you to grab a load of work-space off the stack, and put a pointer to it in another address register. This is used to allocate space for all "local" variables by things like C compilers - so you need a lot of stack-space on Amiga!







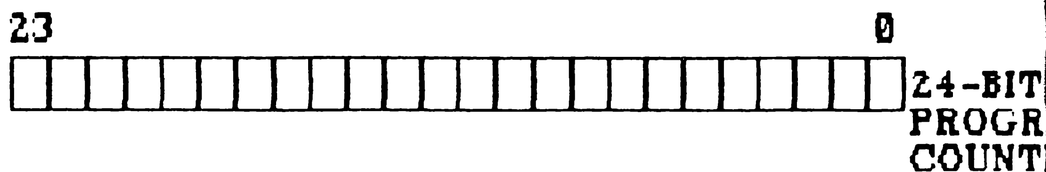


Table 1 - 68000 Registers

ADDRESSING MODE	GENERAL FORM	EFFECTIVE ADDRESS	EXAMPLE
INHERENT		NONE or INHERENT	RTS
REGISTER	Dn/An/SR/ CCR/USP	SPECIFIED REGISTER	SWAP D2
IMMEDIATE	#data	INSTRUCTION or EXTENSION WORD	CMP #$1234,D5
ABSOLUTE	addr	SPECIFIED ADDRESS	MOVE $4321,D3
ADDRESS REG. INDIRECT	(An)	CONTENTS OF SPECIFIED ADDRESS REGISTER	CLR (A3)
ADDRESS REG. INDIRECT WITH DISPLACEMENT	d16(An)	CONTENTS OF ADDR. REG.+DISPLACEMENT	NEG $200
ADDRESS REG. INDIRECT WITH POSTINCREMENT	(An)+	CONTENTS OF ADDR. REG. BEFORE INCREMENTING	ADD (A2)+,D2
ADDRESS REG. INDIRECT WITH PREDECREMENT	-(An)	CONTENTS OF ADDR. REG. AFTER DECREMENTING	SUB -(A1),D6
ADDRESS REG. INDIRECT WITH INDEX AND DISPLACEMENT	d8(An,i)	CONTENTS OF(ADDR REG+INDEX REG.) + DISPLACEMENT	OR 5(A2,D3),D5
PROGRAM COUNTER RELATIVE WITH DISPLACEMENT	label	PROGRAM COUNTER VALUE + OFFSET	BNE FRED
PROGRAM COUNTER RELATIVE WITH INDEX AND DISPLACEMENT	label(i)	PROGRAM COUNTER VALUE + OFFSET +INDEX REG.	OR JIM(A6),D1

Table 2 - 68000 Addressing Modes

MNEMONIC	DESCRIPTION
ABCD	ADD DECIMAL WITH EXTEND
ADD	ADD
AND	LOGICAL AND
ASL	ARITHMETIC SHIFT LEFT
ASR	ARITHMETIC SHIFT RIGHT
Bcc	BRANCH CONDITIONALLY
BCHG	BIT TEST AND CHANGE
BRA	BRANCH ALWAYS
BSET	BIT TEST AND SET
BSR	BRANCH TO SUBROUTINE
BTST	BIT TEST
CHK	CHECK REGISTER AGAINST BOUNDS
CLR	CLEAR OPERAND
CMP	COMPARE
DBcc	TEST COND.,DECREMENT & BRANCH
DIVS	SIGNED DIVIDE
DIVU	UNSIGNED DIVIDE
EOR	EXCLUSIVE OR
EXG	EXCHANGE REGISTERS
EXT	SIGN EXTEND
JMP	JUMP
JSR	JUMP TO SUBROUTINE
LEA	LOAD EFFECTIVE ADDRESS
LINK	LINK STACK
LSL	LOGICAL SHIFT LEFT
LSR	LOGICAL SHIFT RIGHT
MOVE	MOVE

Table 3 - 68000 Mnemonics

MNEMONIC	DESCRIPTION
MOVEM	MOVE MULTIPLE REGISTERS
MOVEP	MOVE PERIPHERAL DATA
MULS	SIGNED MULTIPLY
MULU	UNSIGNED MULTIPLY
NBCD	NEGATE DECIMAL WITH EXTEND
NEG	NEGATE
NOP	NO OPERATION
NOT	ONE'S COMPLEMENT
OR	LOGICAL OR
PEA	PUSH EFFECTIVE ADDRESS
RESET	RESET EXTERNAL DEVICES
ROL	ROTATE LEFT WITHOUT EXTEND
ROR	ROTATE RIGHT WITHOUT EXTEND
ROXL	ROTATE LEFT WITH EXTEND
ROXR	ROTATE RIGHT WITH EXTEND
RTE	RETURN FROM EXCEPTION
RTR	RETURN AND RESTORE
RTS	RETURN FROM SUBROUTINE
SBCD	SUBTRACT DECIMAL WITH EXTEND
Scc	SET CONDITIONAL
STOP	STOP
SUB	SUBTRACT
SWAP	SWAP DATA REGISTER HALVES
TAS	TEST AND SET OPERAND
TRAP	TRAP
TRAPV	TRAP ON OVERFLOW
TST	TEST
UNLNK	UNLINK

Table 3 - 68000 Mnemonics (continued)

Appendix 2 - More about Memory

The situation as regards memory on the original Amiga 1000 was fairly straightforward. The A1000 came with 512K of internal "chip memory" accessible by both the 68000 and the PAD (you could get a system with only 256K, but there wasn't much point); if you wanted to expand on this you could add up to 8 megabytes of external "fast memory" (also known as expansion memory), accessible by the 68000 only, and therefore not subject to cycle-stealing by the PAD. This situation has become confused since the release of the A2000 and the A500, by the arrival of a new form of memory generally known as "slow memory" - it is probably worth trying to explain this, though please feel free to ignore this section if this is your first reading!

The situation on the Amiga 1000 was that available chip memory was checked by Exec on power-up; Exec would then swipe some of this memory for its own use for things like Exec library structures and the system supervisor stack, which was put at the top of chip memory from $07 E800 up to $08 0000. The rest of available memory was put by Exec into a "free memory list", ready for allocation by anything else that wanted it.

Later on, the system would scan for expansion memory, using a complex protocol looked after by a special library called "expansion.library". This would interrogate any add-on cards, looking out for expansion memory (amongst other things); if found, this memory would be linked into the memory free list as fast memory, at the next available location somewhere between $20 0000 and $A0 0000.

From then on, memory allocation was looked after by two Exec routines called AllocMem() and FreeMem(), or by higher level routines built on these such as Exec AllocEntry() and FreeEntry(), or Lattice malloc() and free(). Exec AllocMem() is called with two parameters, the first indicating how much memory is needed, and the second indicating various options, including what sort of memory is wanted - chip memory, fast memory, or don't-care-fast-if-available. This causes a block of memory to be removed from the free list, until released by a suitable call to FreeMem().

This was a nice versatile system; the only problem with it was that it wasted some chip memory on Exec library structures and supervisor stack, which didn't really need to be there and which took memory which could otherwise be used for Intuition screens, graphics structures, digitised waveforms, etc. Since it rapidly became apparent that chip memory was very much at a premium on Amiga, this scheme was modified somewhat on the A2000 and the A500, by adding a new form of memory, now generally known as "slow memory". (To confuse matters, slow memory was once known as "ranger memory", while recent documentation tends to refer to slow memory as "fast memory", while referring to real fast memory as "expansion memory" - we shall ignore this.)

Slow memory is an additional 512K of RAM, built into the A2000, or available as an optional internal RAM-pack (together with real-time clock/calender) on the A500. This memory maps in up at the top, in an area previously reserved for IO etc starting at $C0 0000, thus bringing the Amiga A2000 and A500 total RAM up to a theoretical maximum of 9 megabytes. Slow memory is checked for by Exec BEFORE it checks chip memory at power-up; if found, slow memory is used for things like ExecBase and the supervisor stack, instead of these being put into chip memory. The rest of slow memory not used for these structures is put into the free memory list; from then on it is treated by the system exactly like fast (ie expansion) memory.

The good news is that this gives you the maximum possible amount of free chip memory on the A2000 and A500. The bad news is that slow memory - as the name implies - isn't real fast memory; despite the fact that it lives high up in the memory map, slow memory is in fact on the same side of the gate in the Amiga bus as the PAD. This means that slow memory access suffers from cycle stealing when the PAD is handling high resolution or using a "nasty" blitter, despite the fact that slow memory cannot actually be accessed by the PAD (at least with the current chip set!). Be warned therefore that a program which uses high resolution or a lot of colours, or which does a lot of "nasty" blitting, will not run as fast in a one megabyte A2000 or A500 as it will in a system with real fast (expansion) memory.

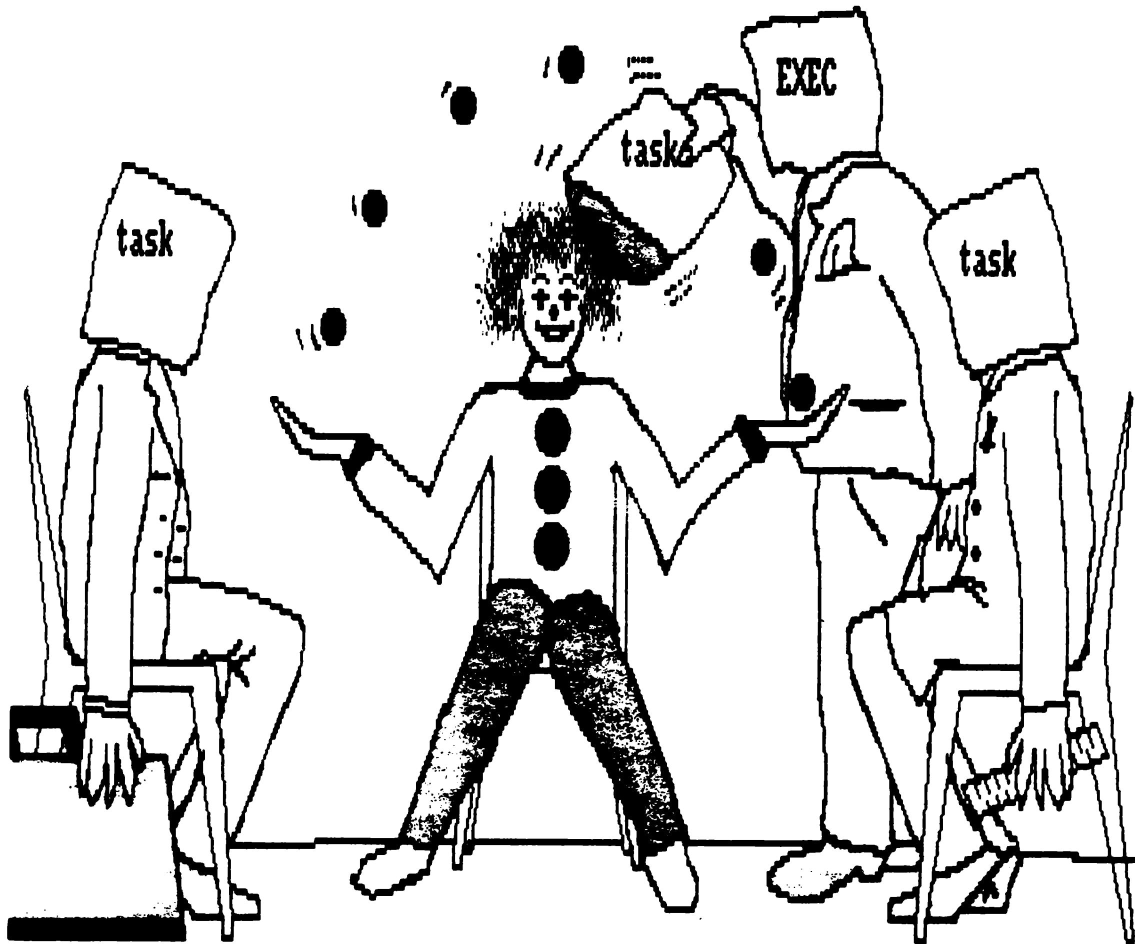
There are two further points worth making, relating to two utilities provided on the 1.2 disks, called NoFastMem and SlowMemLast. The first is fairly simple: some early games programs tended to assume that any memory they found in the machine was chip memory, and they therefore won't work properly on a system with over 512K. To get round this, run NoFastMemory (by double clicking on the icon), which will go through the system allocating any memory that isn't chip memory so that these games will run properly; double click on the icon again to get the extra memory back.

The second point is more subtle: if you have a system with slow memory, then even if you add real fast (expansion) memory, this will tend not to get used as much it should be. This is because Exec links the slow memory into the system free list BEFORE expansion.library links in the fast memory, which means that any remaining slow memory will always tend to get allocated before the real fast memory gets a look in. The solution is to run the program SlowMemLast, which will adjust the links in the free memory list so that slow memory is at the end of the list, so that it will get used AFTER any real fast memory. If you are adding expansion memory to an A2000, or to an A500 to which you have already fitted slow memory, we suggest putting SlowMemLast in your standard Workbench startup sequence.

Part II - The Kickstart Guide To Amiga

Amiga Exec

How To Do Several Things At Once While Doing One Thing At A Time



Exec illustration by Tris Murray.

Section 1 - Amiga Exec

How To Do Several Things At Once While Doing One Thing At A Time

Viewed in hardware terms, what makes the Amiga special are the clever chips, which are able to maintain a high quality colour display with very little effort on the part of the CPU. Viewed in software terms, what makes the Amiga special is the multi-tasking, which is handled by a crucial bit of system software called Exec.

These two areas - the clever hardware and the clever software - are in fact very closely connected. It is because you don't have to tie up the CPU looking after the display all the time that you can afford to use more sophisticated structures and concepts (with higher overheads) in the system software; it is because you can use these structures and concepts that you can do multi-tasking in a reasonably civilised manner.

However, this software sophistication can be a bit of a problem. If you are just out of a computer science degree - or if you happen to have spent the last N years working on Unix systems - then many of the concepts behind the Amiga should be quite familiar. If on the other hand you came to software development from some other background, and thence to 8-bit machines like the Commodore 64, then these ideas won't necessarily be familiar, and you won't find the documentation all that helpful, as it assumes you know them.

For this reason, this book aims to tackle the Amiga from a different angle, from the point of view of people (like ourselves) who know the chips like the 6502 and machines like the 64 pretty well, but who tend to go a bit green when someone says "round-robin scheduling" or "pre-emptive time-slicing". If you are coming onto the Amiga from something like the 64, we hope you will find this useful. If on the other hand you are coming to the Amiga from something like Unix you may find this less useful - if so, you can amuse yourself spotting our errors - please write and tell us!

An introduction to multi-tasking

Multi-tasking on the Amiga is essentially a clever trick pulled on the interrupts. Thus in order to understand how it works, you have to know a bit about 68000 interrupts on the Amiga. We will approach this by first reviewing 6502 interrupts on the 64, and suggesting how you might use them to implement a simple form of multi-tasking. We will then discuss why this would be a pretty silly thing to do - though don't let us stop you of course - then go on to discuss how it can be done in a more sensible way on the Amiga.

Multi-tasking on the 64?

As is now widely known, the 6502 has two interrupt lines - interrupt request IRQ and non-maskable interrupt NMI. If either of these lines is pulled low by external hardware, then the 6502 is forced to perform an interrupt; this means that it finishes the instruction in progress, then saves its current program counter and status register on its stack, sets the "interrupt disable" flag in its status register, then jumps to an address held in a "vector" at the top of memory. This invokes an "interrupt servicing" routine, which typically saves off the registers, does its business, restores registers, then returns from interrupt (RTI). RTI causes the program counter and status register to be restored from the stack (this has the side-effect of clearing the interrupt disable flag); the interrupted program then carries on as if nothing had happened.

IRQ and NMI differ in that IRQ can be disabled by setting the interrupt disable flag, usually using the SEI instruction. This causes any further IRQs to be ignored until interrupts are re-enabled, usually by a CLI (clear interrupt disable) instruction, or by a return from interrupt. NMI (non-maskable interrupt) cannot be disabled, and can be considered as being at a higher priority than IRQ; an NMI can interrupt an IRQ interrupt handler, but an IRQ will not usually be able to interrupt an NMI.

There is a third form of interrupt on the 6502 known as a "software interrupt", in the form of the BRK instruction. When the 6502 hits BRK op-code ($00), it behaves exactly as if it had received a hardware IRQ, but with a special flag set in the status register so that the interrupt handler routines can tell the two apart. BRK is usually used for debugging, eg to cause an entry to a monitor such as Supermon.

On the 64, NMIs and IRQs can each result from a variety of sources, which have to be identified by the interrupt handler routines. However, under many circumstances the only interrupt that needs to be worried about is a "clocked" IRQ, generated every 1/60 seconds by a timer on one of the CIAs. The principal activities caused by the default interrupt handler for this IRQ are to update the clock locations used by BASIC TI and TI$, to update the location used by BASIC stop-key checking, and to scan the keyboard and store any key presses in the keyboard queue.

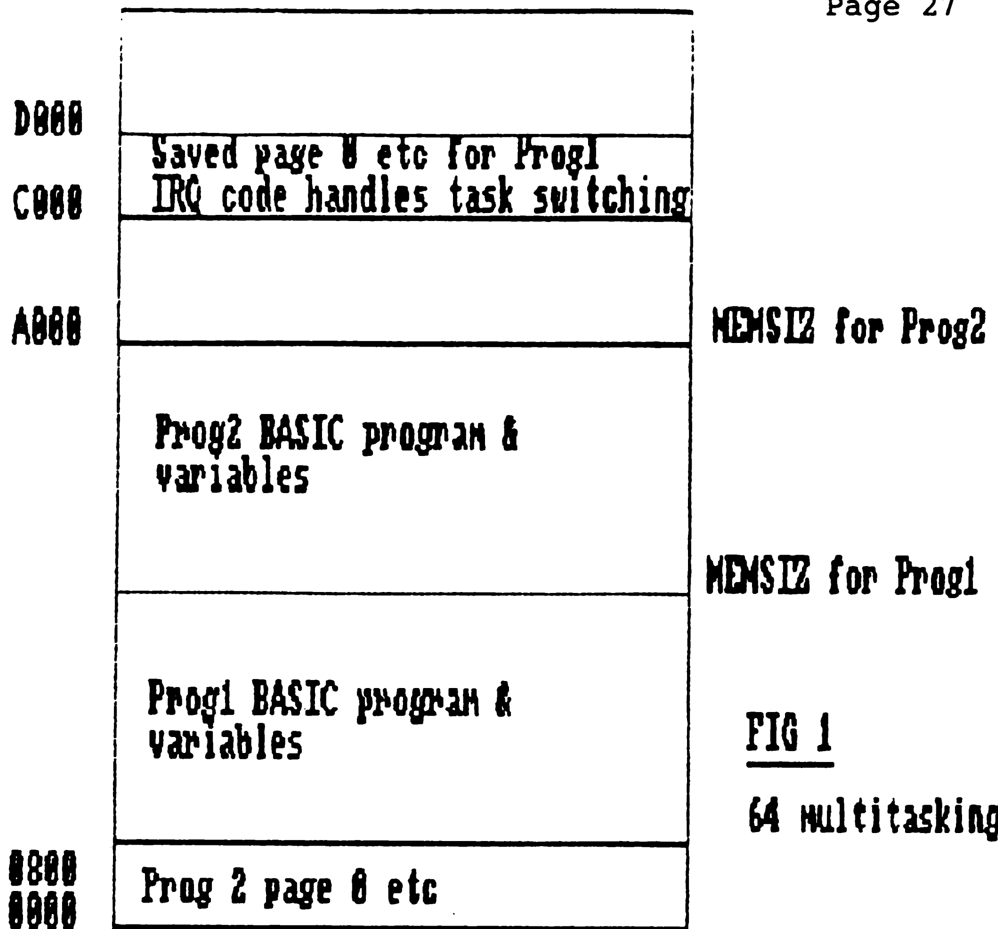
As most 64 programmers are now aware, it is possible to enable other sources of interrupt; for example the VIC chip can be made to cause an interrupt when the electron beam reaches a specified point on the screen, allowing various "split screen" tricks to be implemented, such as changing background, or increasing the apparent number of sprites. In order to do this, it is also necessary to modify or replace the default interrupt handler routines; this can be done quite easily. Other tricks can be pulled just by modifying the interrupt handler; an example is "polling" an external device such as a modem chip, and performing input or output "on the interrupts" if necessary.

Less well known is the fact that you can in fact spend just about as long as you like before "returning from interrupt" without upsetting the 64. For example, it is possible to run a "snapshot" utility on the interrupts, which allows a screen dump to be made at any point during the execution of a BASIC program, following which BASIC execution will continue. This works by modifying the interrupt handler to check for some special key combination; if found interrupts are re-enabled and a screen dump routine invoked. This can then run perfectly normally; indeed it has no way of knowing that it is actually running "on the interrupts" (the stack pointer is a bit lower than it would otherwise be, but what the heck). The screen dump has to take care to save and restore any locations used in page zero etc, and to use workspace separate from BASIC for its own variables; when it has finished it can then pull registers from the stack and RTI, causing BASIC to resume as if nothing untoward had happened.

(This utility gets into trouble if BASIC happens to be in the process of using the printer when the snapshot is invoked. This is an example of the dreaded contention, of which more anon.)

It would theoretically be possible to extend this technique in order to provide at least a limited form of multi-tasking on the 64 in BASIC. In order to do this, you would do "task switching" on the interrupts by saving off BASIC work-space (page 0 etc) somewhere private (say around $C000), then setting the pointers appropriately for another BASIC program, which would have its own work area, variables etc somewhere else in memory - say above the value of MEMSIZ for the first program. BASIC could then be kicked off again and the second program run for a while; a few interrupts later you could then restore the first program's pointers then restore registers and RTI; the first program would then carry on as if nothing had happened. This is a simple form of multi-tasking - see Fig 1.

Limited versions of multi-tasking are in fact available on some eight-bit micros, but there isn't really much point. For one thing you tend to run out of memory; for another, the overhead in saving everything off and restoring it as suggested is rather high, so you tend to spend so much time "task switching" that you don't actually get time to do anything useful. In order to make the whole business practical you need more memory, a faster more versatile processor, and preferably some clever chips to look after the screen without involving the processor too much; in fact an Amiga will do very nicely.



EXEC

Fig 2

exec task ready
queue

2-task Amiga

task
control
block
(ready)

task
control
block
(running)

address of cleanup code
other stuff
task's saved registers

task
user
stack

address of cleanup code
other stuff

task
user
stack

(cleanup
code)

task
code

task
data

(cleanup
code)

task
code

task
data

Amiga Exec

Exec is a collection of routines at the "lowest level" of the Amiga; it is used by all the other bits such as graphics libraries, Intuition, AmigaDOS, device drivers etc, and can also be used directly by application programs. It is also in charge of interrupts on the machine, and as such exercises a controlling influence over everything else. The principal thing looked after by Exec in its controlling role is a sophisticated version of multi-tasking - the Amiga can run several programs at once, each of which thinks it has a "virtual machine" all to itself. It does this by means of task switching on the interrupts, using some special facilities of the 68000.

Enter the 68000

In general, if you are familiar with the 6502, you shouldn't have very much trouble understanding the 68000. While the two chips are only distantly related, and are organised internally in a very different way, they are in many respects conceptually quite similar. In particular, both use a small number of instructions with a large number of address modes; the 68000 has about 56 fundamental instructions (quite a lot of which come in various flavours such as byte, word and long-word) and eleven addressing modes. However, the 68000 does have a number of features not found on processors like the 6502, which come in handy when it comes to multi-tasking.

The 68000 doesn't have anything equivalent to the 6502 "page zero". Instead it has eight internal 32-bit data registers and eight 32-bit address registers. The latter can be used for a wide variety of indirection, such as

```
JSR  (A6)      - call a subroutine whose address is in A6
```

or even

```
JSR -6(A6)     - call a subroutine whose address is 6 bytes
                 less than what's in A6
```

Address register A7 is reserved for use as the stack pointer. Since this is a 32-bit register, this means you can have a stack as big as you like anywhere in memory. (It also means that an exploding stack can completely smash the machine, but there you go.) The stack tends to be used very heavily on the 68000. For example, the LINK instruction can be used to grab some temporary work area off the stack and put a pointer to it in one of the other address registers; this area can then be accessed by suitable indirect addressing. This technique is used by most compilers (including Lattice C) to allocate storage for all local ("automatic") variables, so you need a lot of stack space!

3. The 68000 has three interrupt lines, which are used together to provide eight priorities of interrupt. Instead of an interrupt disable flag, it has a three-bit interrupt mask in the status register; this is used to prioritise interrupts so that a level 1 interrupt can itself be interrupted by levels 2 to 7, but not by levels 0 or 1. The number of priorities is effectively increased by one of the Amiga custom chips, the 4703 (or "PAULA"); this watches fifteen possible sources of interrupt (NMI, copper, expansion bus, disk , serial i/o, audio channels, blitter, vertical blank, etc), and decides if and when to interrupt the CPU and with what priority.

   The full details of interrupt handling on the Amiga are fairly complicated, especially when it comes to the details of the interaction between PAULA and the 68000. However, the overall effect is not too dissimilar to the 6502 case discussed above:

   1. PAULA gets an interrupt from some bit of hardware. It flags which interrupt was requested by setting a bit in one of its registers, then checks another register to see if this interrupt is enabled. If so, it generates an interrupt to the 68000 at the appropriate priority.

   2. If the 68000 is already servicing an interrupt, it checks to see if the new priority is greater; if not it ignores it for the time being. Otherwise it switches into "supervisor mode" (see below), saves off status register and program counter on the system stack, sets its interrupt mask appropriately, then jumps by means of a vector in the bottom 1K of memory to an appropriate entry point in Exec. Exec then further decodes what is going on by looking at PAULA's registers, and calls the appropriate interrupt handler.

   3. On return from interrupt (RTE), the 68000 restores status register and program counter; this has the side-effect of restoring the interrupt mask to its previous level. It then restores the mode from which it was interrupted (this is usually "user mode" - see below), and exits.

   The 68000 also supports a wide variety of software interrupts, including various error conditions like illegal instructions or divide-by-zero, and 16 TRAPs, which can be used to initiate special processing in a way similar to 6502 BRK. Again, these are generally used for purposes like debugging; the system default action on traps is to give you a guru number to meditate on, then (optionally) to sling you into ROMWACK.

   Now, there is a problem with terminology here. On a 6502, we usually talk about hardware interrupts (IRQ and NMI), and software interrupts (BRK). Motorola on the other hand don't talk about interrupts at all - instead their documentation refers to externally and internally-generated exceptions, which are just like interrupts, but more wonderful. Amiga, just to be different again, tend to talk about hardware interrupts and software traps; they use the terms "exception" and "software interrupt" to

describe two tricks of their own, which are touched on below. This is very confusing. From now on, we will try and use the Amiga terminology in order to be consistent with the documentation - okay?

4. The 68000 can work in two modes, known as supervisor and user modes, distinguished by a bit in the status register. Generally speaking, user programs (this means you) run in user mode; supervisor mode is only entered if the 68000 gets an interrupt or trap. The most significant difference between the two modes is that each has its own private stack pointer; Exec can therefore run "on the interrupts", with the benefit of its own private system stack. The only other difference between the two modes is that a few instructions are "privileged" and can only be executed in supervisor mode; this includes all operations affecting the "system" part of the status register, so you can't barge into supervisor mode from user mode directly; it has to be entered legally. An attempt to use a privileged instuction from user mode generates a trap - on the Amiga, this will usually give you guru number eight.

68000 multi-tasking

A number of special instructions exist on the 68000 to allow the user stack pointer to be manipulated from supervisor mode (it is NOT possible to access the supervisor stack pointer from user mode!). The significance of this is that it is possible to have several different user programs ("tasks") in memory, each with its own private stack; Exec can then swap between tasks "on the interrupts" by fiddling about with the user stack pointer.

As mentioned above, there are many sources of hardware interrupts on the Amiga. The one of immediate relevance in understanding how the machine does multi-tasking is the vertical blank; this is generated once for every scan of the video display, and can be thought of as the Amiga's closest equivalent to the 64's "clocked" IRQ. (Coincidentally, it also happens about every 1/60 seconds in the USA, or every 1/50th elsewhere).

Thus as an application program on the Amiga, you will be running as a task somewhere or other in memory, with the 68000 in User mode, with your own data areas, and your own user stack. Every vertical blank, Exec will be waking up and having a look at you, running in Supervisor mode, with its own private supervisor stack. If Exec decides to leave you unmolested, it will simply return from interrupt and let you carry on.

If on the other hand, Exec decides to let some other task have a go, it will save the current values of ALL your registers on your (user) stack, then remember what your user stack pointer was in something called your "task control block". It will then restore some other task's user stack pointer, pull the last saved values of its registers from its user stack, then return from interrupt. The other task will then carry on as if nothing had happened.

In fact, Exec thinks about multi-tasking at the end of any form of interrupt processing, not just vertical blank. It can also be forced to think about it by other means, for example by a task calling the Wait() function, which indicates that it doesn't want to run for a while.

Memory management

In order for a variety of tasks to run independently as described above, it is very important that they are not allowed to interfere with each other, eg by trying to use the same memory. Thus memory allocation has also to be looked after by Exec, which keeps lists of what regions in memory are currently free, and what regions are allocated.

There are two aspects to this, the first of which is looked after by AmigaDOS, and the second of which is up to the programmer. An AmigaDOS program file is stored as a number of "hunks" of code and data, each of which has associated with it some relocation information, allowing code hunks to be put anywhere in memory. This is handled by the AmigaDOS scatter-loader, which asks Exec to allocate memory for each hunk, then loads, relocating as it goes. AmigaDOS also asks Exec for memory for the task's stack - the amount of memory allocated for the stack is picked up from the .info file if the program is run from Workbench, or controlled by the current setting of STACK from the CLI. Thus a program can rely on internal code and data, and on its private user stack being allocated to it by AmigaDOS; this memory remains allocated until the program terminates.

The program will require further memory for buffers, bit-maps and whatever structures it cares to create. The allocation of memory for this is handled by various routines in Exec, the simplest of which is AllocMem(), which looks for a block of free memory of the size and type requested, and returns a pointer to it if found. This means that any memory allocated in this way must be addressed indirectly; if you are using something like a C compiler this is very straightforward (use pointers!).

Contention

There are other ways of getting into trouble in a multi-tasking environment besides problems with memory. An example is handling a hardware resource like the parallel port or the blitter; it is possible for one task to start an operation on a hardware device, then for another to be cut in by Exec and try to do something quite different, resulting in system confusion.

This problem is handled on a resource by resource basis by the associated system software; it is not looked after directly by Exec. In the case of the parallel port, access to the device driver is obtained via the Exec call OpenDevice(), which can pass

the device a flag requesting exclusive access; OpenDevice() in turn tries to open a lower-level entity called a resource which is directly concerned with granting or refusing access to the parallel port hardware ("misc.resource"). If another task has exclusive access, then the attempt to open the resource will fail, and other tasks calling OpenDevice() will return an error until the first task has finished. In the case of the blitter, it is possible to claim exclusive use using a graphics library routine OwnBlitter(), or to queue a non-exclusive request using QBlit(). These different cases will be considered in more detail later.

Note that if all else fails, it is possible to stop Exec from task-switching for a while using two routines Forbid() and Permit(), or even to switch off interrupts completely using Disable() and Enable(). It should not be necessary to do this except in exceptional circumstances however.

Time-slicing

The process of Exec deciding which task should be running, and getting it going if necessary, is called task scheduling and dispatching; the mechanism used to do it is called pre-emptive time-slicing.

Consider a rather boring Amiga (only twice as interesting as its competitors) which is only running two tasks; assume these tasks are quite independent of each other, and are of equal "priority" (see below). In this case, Exec will simply task swap as described above at pre-set time intervals (time-slices); this time interval, which is known as a "quantum", is currently set to four vertical interrupts, or 1/15s. This time-slicing is "pre-emptive" in that the task losing the processor doesn't get any say in the matter; from its point of view it is just as if it had an abnormal very long interrupt. See Fig 2.

More than two tasks of equal priority are handled in a similar manner, with Exec switching the processor between them every 1/15 seconds, with each task taking its turn in a "round robin" fashion.

Task priority

Tasks are added to the system by setting up a task control block somewhere in memory, then calling a routine called AddTask with the address of this structure, a "kick-off" address for the task, and an optional "clean-up" address, specifying what to do if your task decides to RTS from its entry stack-level for some reason. The task control block contains various information for the task such as a name, the upper and lower bounds for the task's stack, and an initial stack pointer; it also contains a single byte interpreted as a number from -128 to 127 for the task's priority.

When first experimenting with tasks, it is a good idea to always set the priority to zero; this is a safe "neutral" value. It is however possible to choose lower or higher values. The rule used by Exec to handle priority is very simple; if a high priority task wants to run, a lower priority task will never get the processor; Exec will only time-slice between tasks of the same priority. This ceases to be the case if the high priority task indicates that it doesn't want to run for a while by entering a "wait" state (see below); lower priority tasks then get a look in. Since the Amiga uses tasks for a lot of system activity such as most I/O, this means that high-priority tasks should be used only when necessary and then with caution. System tasks usually have priorities between -20 and +20.

At this point, it may be worth considering some real tasks running in the Amiga. If you open a CLI window in the workbench, then invoke OldWack, you can get a list of tasks by typing TASKS. This will be something like the following:

Type	Priority	Status	Name	
Process	0	run	Background CLI	(you using WACK)
Process	0	wait	CLI	(CLI process)
Process	5	wait	CON	(CLI console device)
Process	10	wait	File System	(CLI filing)
Process	1	wait	Workbench	
Process	1	wait	File System	(Workbench filing)
Task	5	wait	trackdisk.device	(CLI disk device)
Task	5	wait	trackdisk.device	(CLI other drive)
Task	20	wait	input.device	(mouse/keyboard/timer)
Task	5	wait	trackdisk.device	(workbench disk device)
Task	5	wait	trackdisk.device	(workbench other drive)
Process	5	wait	RAW	(WACK RAW console)

"Processes" and "tasks" are distinguished by "node types" of 13 and 1 respectively. A task is an Exec concept as discussed in this document. A process is an AmigaDOS structure built on the idea of a task; it consists of a task control block, a "message port" (see below), and a lot of other stuff.

Waits and signals

Tasks operating independently of each other as discussed above are not terribly exciting; things become more interesting when it becomes possible for tasks to communicate with each other, eg for one task to send another a "message" asking it to do something, or for a task to go to sleep (give up the processor) until it is woken up by some action on the part of another task. The process of tasks going to sleep and waking up is handled at the lowest level by Exec using a mechanism called signals.

Tasks on the Amiga can be in three principal states, which are as follows:

```
RUNNING - I've got the processor
READY   - I want the processor
WAITING - I don't want the processor until so-and-so happens
```

The current status of each task is flagged by Exec within the task control block. In addition, Exec maintains its task control blocks in two lists, a "ready queue" ordered on task priority, and a list of waiting tasks in no particular order. A task indicates that it wants to go to sleep until something external to itself happens by calling a general-purpose Exec routine Wait(), or a more special-purpose routine such as WaitPort() (see below). This causes Exec to remove the task's control block from the READY queue and put it in the WAITING list; it then returns to time-slicing between the tasks at the front of the READY list, i.e. those of highest priority.

It should be pointed out that a lot of tasks will spend most of their time waiting. An example is a task concerned with disk i/o; this will spend most of its life waiting until some other task requests disk activity. This is a pretty good idea, since tasks in wait states don't tie up the processor.

The reverse process - of getting a task out of the WAITING list and back into the READY queue - is handled by a mechanism called signals. Each task has associated with it 32 signal bits; the low order 16 of these are reserved for system use, while the high order 16 are free for whatever you want to do with them. In fact, each task control block contains four long-words (4 * 32 bits) of signal-bit information, flagging which signal bits have been already allocated for use by this task, which signal bits the task is currently waiting for, which signals have been received, and which signals should cause a special form of processing called an "exception" (not to be confused with what Motorola mean by an exception - aarrrgghhh).

Signal-bits are most often used in conjunction with a higher level inter-task communication mechanism called messages and ports. In this context, they are usually allocated for you by Exec; however you can just as well look after them yourself. The meaning of each bit is up to you; for example you might want to have one signal bit flagging messages coming in from the console, another flagging messages from the disk device, and a third connected to a timer. The way to use signal bits directly is as follows:

1. The safe way to claim a signal bit for some purpose is to call an Exec routine called AllocSignal(). If called with an argument of -1, this will return the number of the next free bit to you from 16 to 31 (you will have to convert this into a bit mask), and flag that bit as allocated.

2. To go to sleep until some event (or a choice of several events) of interest takes place, call Wait() with an argument which is a bit-mask indicating which signal (or signals) you are waiting for. Exec will then put you in the WAITING list until something happens to set these bits.

3. When something does happen, Exec will put you back in the READY queue so that you again have the chance to run. When you get the processor again you will return from Wait(); the value returned indicates what signal (or signals) happened to cause you to wake up again.

4. As mentioned above, signals are most often caused by "messages" arriving at "message ports". However, it is possible for one task to signal another directly, by calling an Exec routine Signal(task,mask), which sets the signal bits specified in the control block of the task indicated.

Messages and ports

Simply waking up another task by calling Signal() is of limited use; you usually need to send the task some data as well - for example, you might want to output some text by sending a string to the console device. This is handled by sending messages to message ports.

A message port is a data structure linked to a task control block. In order to do any form of I/O at all, a task needs at least one message port, and it is frequently convenient to use several. Each port has linked to it a queue of messages from other tasks. The arrival of a message at a message port usually causes the associated task to be signalled; this causes it to wake up (return from a wait state) and do something about the incoming message. Once it has finished processing a message, a task usually needs to let the task that sent the message know that it has finished with it. It does this by replying the message by sending it back to the task that originated it; it is able to do this because each message contains a long-word which is either the address of the port to reply to, or zero if no reply is required.

Let's take that again, slowly. Suppose we have an application where we have two tasks, a "main" task and a "child" task, which we want to be able to communicate in a simple way, by the main task sending the child task messages. This can be handled roughly as follows - a detailed example is given in C later.

1. Both tasks need to get going somehow. This can be done by being kicked off as a process from AmigaDOS; alternatively a task can spawn another task using the Exec-support routine CreateTask().

2. In order to be able to communicate, both tasks now need message ports and associated signal bits. Creating a message port, linking it with the associated task control block and allocating a signal bit to flag arrival of messages can all be handled by calling an Exec-support routine CreatePort().

3. In order to be able to send messages to each other, the tasks need to know where in memory to find the other task's message port. There are two ways of doing this.

   a. If the two tasks are closely collaborating, they will probably be compiled and linked as part of the same program. If so, they will know where each others' message ports are anyway - these are called "private ports".

   b. If the two tasks are not part of the same program, then they have to use "public ports". A public port must be given a name; it can then be added to a list of ports maintained by Exec, using the Exec routine AddPort(). This will be handled for you if you use the Exec-support routine CreatePort(); if passed a non-null name, this routine assumes a public port, and calls AddPort() accordingly. Once this has been done, another task can find the port by using the Exec routine FindPort(); this causes Exec to search its list for a specified port-name, and return the address of the port if it finds it.

4. The child task can now enter a wait state, until a signal bit goes to indicate that a message has arrived from the main task. The simplest way to do this is to call a routine called WaitPort(), which does just this. Alternatively, if the arrival of the message is only one of a variety of possible interesting events you want to wait for, then you can call Wait() directly, with a bit-mask which includes the signal bit associated with the message port; if you got Exec to create the port, you can find out which signal bit it allocated by looking at the message port structure.

5. The main task can now send a message to the child task. It does this by allocating memory appropriately (using the Exec routine AllocMem()), then setting up a message structure, followed by the message data. The message structure includes a "reply port" address; the main task fills the address of its own message-port in here. The message can then be sent using the Exec routine PutMsg(). The main task can now get on with something else, and/or start watching its own message port for a reply, probably by entering a wait state using WaitPort().

6. Once the message has been sent, the child task will be signalled, and will return from WaitPort(). The message can then be removed from the queue using another Exec routine GetMsg, which returns the address of the first message in the queue, or zero if there are no more messages.

7. The child task can now examine the message, and take appropriate action. When it has finished with it, it should return it to the specified reply port; this can be done conveniently using an Exec routine ReplyMsg().

8. In the general case, there may be more than one message queued at the message port, despite the fact that the child task was only signalled once. If this is a possibility, then the child task should continue calling GetMsg() and processing any further messages, until GetMsg() returns zero. The child can then return to a wait state using WaitPort().

9. When the main task gets the reply from child task, it should remove it from its message port using GetMsg(). Note that it is possible for the child task to pass data back to the main task by modifying the message data before replying; if so the main task can now make use of the returned data. It can then de-allocate the memory used for the message, use it for another message, or whatever.

See fig 3 for an illustration showing the relation between tasks, message-ports and messages.

There is a final subtlety to this business which is well worth noting. This is that very little actually gets moved about in memory when a message is sent; the message data actually stays in the same place, but gets attached to the child's message port by cunning use of pointers. For this reason, the main task must be very careful not to touch the message data, or de-allocate the message memory etc, **until the child task has replied the message.** Another way of looking at this is to say that by sending the message, the main task grants the child task a temporary licence to mess about with a bit of main task's memory; by replying the message, the child task returns this memory to the main task.

another message port
task's message port
task's message port
task control block
signal bits
message
task sending message
queue of messages
message
reply-port
task stack
task code
message
task stack
task code

Figure 3 - Messages and Ports

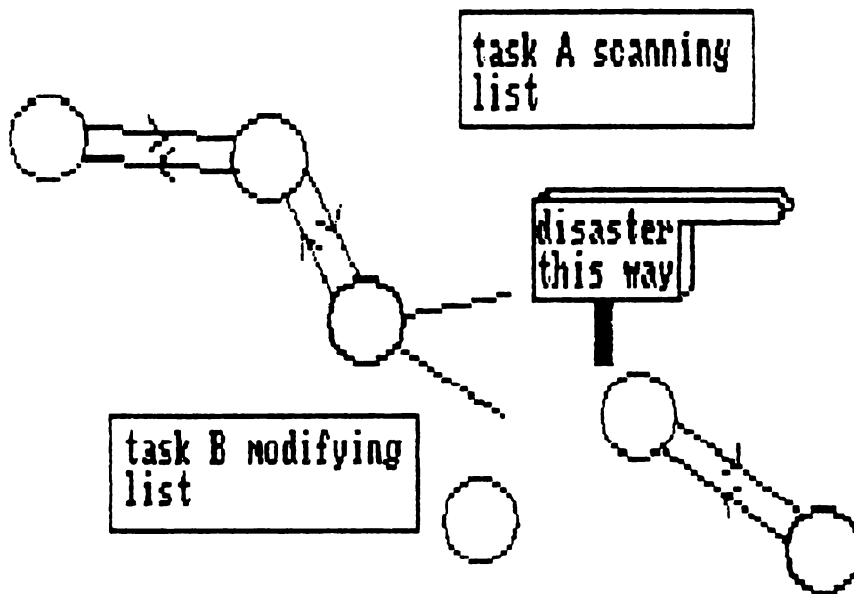


Figure 4 - Two Tasks Contending!

Limitations of tasks

All of this sounds great until you try to use it and the Amiga crashes with a guru (whoever wrote the example of multi-tasking in the original V1.0 ROM kernel manual had a great sense of humour). The problem is principally connected with AmigaDOS. As mentioned above, AmigaDOS does make use of Exec's structures and routines to handle multi-tasking; however it builds on top of Exec's "task" structure (ie a task control block) to create its own structure called a "process", which is a task control block, followed by a message port, followed by various other stuff connected with what the console is, where its window is, etc. AmigaDOS expects this information to be there, ie it expects to be called from a process; if you call it just from a task you will crash the Amiga! This places very severe restrictions on what you can do in a sub-task:

1. You can't call any AmigaDOS functions directly.

2. You can't use a lot of Lattice functions such as printf(), since these call AmigaDOS.

3. You can't open any disk-resident libraries or devices, such as the icon library or the printer device, unless you are sure they are currently in RAM; otherwise AmigaDOS will try and fetch them from disk, thus crashing the system.

On the other hand, there's no problem at all calling ROM libraries such as Intuition and the graphics libraries, or of opening resident devices such as the console. Thus you can perfectly well spawn a sub-task to handle a bit of animation (say) while your main task gets on with something else; or you might want to have a "pre-processor" task sitting on top of the console, passing stuff onto your main process in some pre-digested form. If you do need to call DOS functions from a sub-task, there are two solutions:

1. Be a sub-process instead. In order to do this, you will probably be compiled and linked separately; you can then be kicked off by AmigaDOS Execute(), or by AmigaDOS LoadSeg() followed by CreateProc().

2. Have a dedicated sub-process (or even your main process) handling the interface to AmigaDOS. Then when a sub-task wants to talk to AmigaDOS, send a message to the dedicated process, and let it talk to AmigaDOS.

A C example of a process kicking off a sub-process is given later.

Interrupts

As indicated above, interrupts on the Amiga are essentially the territory of Exec - you can use them yourself, but you have to ask Exec nicely. As mentioned above, the terminology used when discussing interrupts on Amiga tends to be confusing; a summary of the main concepts is as follows.

1. Hardware interrupts. These are looked after by one of two mechanisms - interrupt handlers and server chains. Interrupt handlers are used by high-priority-copper, disk, serial port, audio channels and "software interrupts". Only one handler is allowed per source of interrupt, and you are unlikely to want to change the system defaults; however, if you must, this can be done using a structure called an "Interrupt", and a routine called SetIntVector(). Server chains are used by NMI, the 8520s, the blitter, vertical blank and the copper; they allow tasks to share interrupts, by calling each routine in the chain successively, allowing (say) a number of tasks to synchronise with vertical blank. It is more likely that you will want to try this - if so, you use the same interrupt structure, and a routine AddIntServer(). Note that the Amiga gets upset if you spend too long on the interrupts, particularly when servicing an interrupt of high priority.

2. Traps. These are a form of 68000 special processing very similar to a hardware interrupt, but caused either by an error condition (eg address error, illegal instruction, divide-by-zero), or by 16 special TRAP instructions. If you want to do your own trap handling, you can set up two pointers in your task control block to point to your trap-handling code and (optionally) to a separate data area - this second pointer is really for the system's convenience, since its trap-handling code may be in ROM. If you don't do this, the system will set up default trap-handling, which gives you a guru number when you get a trap. If you want, you can get Exec to help you with TRAP allocation within your task, by using a routine AllocTrap(), which behaves in a way very similar to AllocSignal() discussed above. Note that there is no problem with different tasks using the same traps - Exec will always pass the trap to the task that is currently running. Traps are used by monitors like Wack to implement things like break-points.

3. Exceptions. This is a cunning Amiga trick to allow a task to have "private interrupts" connected to a signal bit. If you want to use this, you set up pointers in your task control block to point to special exception-handling code and data-area; you then call an Exec routine SetExcept() to indicate which signal bits you want to cause exceptions. You can for example arrange for message arrival at a given message port to cause an exception; you can then go and do something quite different, knowing that as soon as a message arrives your exception code will be invoked.

4. Software interrupts. This is another cunning Amiga trick. "Software interrupts" on the Amiga use the same data structure as hardware interrupts; they run at lower priority than hardware interrupts and traps, but higher than normal tasks. They have two main purposes:

   a. As mentioned above, the Amiga gets upset if you spend too long servicing a hardware interrupt. To get round this, you can arrange for the hardware interrupt to invoke a software interrupt using the Cause() function. The software interrupt will then be processed after the return from hardware interrupt, but before returning to normal multi-tasking. It is also possible to use Cause() from within a task; if so, the task is interrupted immediately, and there will be no return to normal multi-tasking until the software interrupt has finished.

   b. It is possible to set up a message port to cause a software interrupt, instead of signalling a task. Sending a message to this port will then immediately invoke the software interrupt, again at a higher priority than normal multi-tasking.

Note that there are therefore three principal things that you can arrange to happen when a message arrives at a message port:

1. In the normal case, when a message arrives, a task associated with the port is signalled, and processes the message when it gets round to it.

2. If the message is more urgent than this, it can be arranged to cause an exception, in which case the associated task will process it as soon as it is next woken up, even if is currently busy doing something else.

3. If the message is really urgent it can be arranged to cause a software interrupt; this will be executed at once, at a higher priority than normal multi-tasking.

Programming implications

As was stated at the start of this section, Exec does its job very well, so if you're not doing something exotic like using interrupts, you can more-or-less forget the multi-tasking, and just let Exec get on with it. However, there are a few rules you have to keep:

1. You MUST NOT simply hit the hardware when you feel like it - what do you think this is, a 64? Some other task might be in the middle of some delicate operation when you come blundering in - this will cause weird intermittent crashes which will be very hard to track down. If possible, use a system library call instead; if you must access hardware like the blitter directly, do it in a decent manner by first of all claiming it, then accessing it, then releasing again when you have finished.

2. Similarly, be careful with memory allocation. You can make direct references to data actually in your code, as these will be fixed up by the scatter-loader; for all your other needs you should AllocMem(), then access the memory returned to you indirectly.

   You need to be a bit careful with your options when calling AllocMem(). Data structures which are accessed by the special chips should be AllocMem'd MEMF_CHIP; data structures (such as messages) which are going to be accessed by more than one task should be AllocMem'd MEMF_PUBLIC - this is for upward compatibility with any future products which may support hardware memory partitioning. Note that structures like this should NOT be declared implicitly as data in the program. Finally, if you are going to create structures like task control blocks yourself, you should do so MEMF_PUBLIC¦MEMF_CLEAR; however in these cases we would recommend using support routines such as CreateTask(), then adjusting the structures returned if necessary.

3. If you need to wait for something to happen, call Exec Wait() or WaitPort(). "Busy waiting" by wizzing round a tight loop is very bad manners - why tie up the processor doing nothing? - and may cause the system to hang if you are running at a higher priority than the task that you are waiting for!

4. The interrupts belong to Exec; don't mess around with them by directly changing the processor interrupt mask or the bottom-of-memory interrupt vectors. If you want to use interrupts, ask Exec nicely.

5. Finally, be careful about contention. Quite innocent-seeming activities like "bunny hopping" through a system list can cause trouble if some other task is updating the list at the time - see Fig 4. You can frequently avoid this sort of problem by calling Exec routines - such as

FindTask() to search Exec's task lists - rather than doing it yourself. If you MUST access system lists, then use Forbid() and Permit() to disable task switching where necessary.

Structures and lists

So far, this section has deliberately not given details about the exact mechanisms used to maintain lists, task control blocks etc; this has been because we have been trying to concentrate on the principles of what is going on, rather than implementation details that you can find in the ROM kernel manuals. However, it may be worth saying a bit about lists, and giving at least a summary of some important structures used in lists.

Linked Lists

From the discussion above, we might expect that Exec keeps tables somewhere showing which tasks are running and which waiting, what the last saved task user stack pointer was, etc. This is more or less correct, except that the Amiga doesn't use tables for anything much - it uses linked lists.

In a table, items of information ("elements") are ordered implicitly by means of their arrangement in memory - see Fig 5. In a linked list, elements of the list are ordered explicitly; each element of the list contains the address in memory of the next element, i.e. it contains a pointer to it - see Fig 6. The order of the list as maintained by the pointers does not have to correspond with the actual arrangement in memory - indeed the elements could be splattered about all over available memory.

In order to scan a table, you start at the beginning and search sequentially through it until you find what you are looking for. In order to scan a linked list, you have to "bunny hop" through it as follows:

```
DO

     Get address of next element
     Examine this element and do with it as you will

WHILE you haven't run out of list
```

You can tell when you've run out of list, because the pointer to the next element is then zero.

The disadvantages of linked lists is that there is a small overhead due to the pointers (each address takes four bytes), and because the processing of scanning through the list by "bunny hopping" as described can take a bit longer than sequential scanning. The advantages are that you don't have to move everything to insert a new element - you just twiddle the

pointers as shown in Fig 7 - and that it is a "no limits" structure. A table can go on growing only until it fills the space allocated to it; a linked list can grow until it fills the whole of memory.

This sort of arrangement, with "structures" scattered about in memory, containing "pointers" to other structures, is exactly what C was designed to be good at; hence the dominant position of C on the Amiga. However, as regards linked lists, you don't have to write your own routines; Exec contains a number of general purpose routines to handle its own linked lists, and these are of very general usefulness; their use from application programs is recommended.

In fact, there are two subtleties about linked lists as used on the Amiga.

1. The lists are doubly linked, in that each element contains a pointer back to the previous element, as well as a pointer forward to the next element. This makes it possible to bunny hop backward if this is more efficient. Each element in the list starts with a structure called a "Node", which is followed by the actual list data:

    Node: pointer to next node (successor) - 4 bytes
    pointer to previous node (predecessor) - 4 bytes
    node type - 1 byte
    node priority - 1 byte
    pointer to node name - 4 bytes

    Data for list element follows.

    Besides the pointers, the node contains a type used to distinguish between nodes used for different purposes - eg tasks, message ports, messages etc - a priority which can be used to order the list, and a pointer to a node name; in the case of a linked list of task control blocks, the name might be the name of the task or process, eg "Background CLI".

2. The two ends of the list are "tied up" by use of a cunning structure called a "list header" - this structure is simply known as a "List" in order to confuse you. This is arranged as follows:

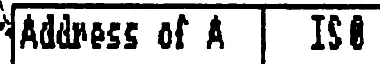
    List: pointer to first node in list
    pointer to first node predecessor = pointer to last node successor = 0
    pointer to last node in list
    list type
    spare byte (padding)

    A complete linked list is illustrated in Fig 8.

Fig 5

Fig 6

Fig 7



List
Pointer to first
Zero marks start/end
Pointer to last
List type
Spare

next
prev.
next
prev.

Node
Pointer to next node
Pointer to previous node
Node type
Node priority
Pointer to node name

Data follows

Node
Pointer to next node
Pointer to previous node
Node type
Node priority
Pointer to node name

Data follows

Node
Pointer to next node
Pointer to previous node
Node type
Node priority
Pointer to node name

Data follows

prev.
next
prev.
next

Fig 8

Task control blocks

Exec maintains its READY queue and its WAITING list as two linked lists of task control blocks, known simply as Tasks. These are arranged as follows:

```
     Task:      Node, includes list pointers, "task" node-
                           type, and pointer to task name.
                Flags - 1 byte
                State - running, ready, waiting etc - 1 byte
                Interrupt disabled nesting - 1 byte
                Task disabled nesting - 1 byte
                Signal bits allocated - 32 bits
                Signals to cause exit from wait - 32 bits
                Signals we have received - 32 bits
                Signals to cause "exceptions" - 32 bits
                Traps that have been allocated - 32 bits
                Traps enabled - 32 bits
                Pointer to data area for "exceptions"
                Pointer to code to handle "exceptions"
                Pointer to data area for traps
                Pointer to code to handle traps (default is "give guru"
                Last saved task user stack pointer
                Address of task user-stack bottom boundary
                Address of task user-stack top boundary
                Routine to call when task is about to lose processor
                Routine to call when task regains processor
                List structure to tack tasks private memory
                         list onto if you want to (up to you)
                Spare pointer (also up to you)
```

A message port also starts with a node; this is to allow Exec to tie public ports into a linked list that it can search when asked to FindPort(). The structure is as follows:

```
     MsgPort:   Node - includes msgport type, and name.
                Flags, including message arrival action
                       flags - signal task, cause software
                       interrupt, ignore message.
                Number of signal bit to use if signalling task.
                Pointer to associated task control block,
                       or software interrupt structure
                List structure, with list of arriving
                       messages attached to it.
```

Messages are then arranged as a list (queue) tied onto the message port:

```
     Message:   Node - ties messages to port.  Message-type
                Address of message port to reply to
                       (null if reply not needed)
                Length of message data in bytes
```

The message data then follows this structure.

The interrupt structure (used for handlers, server-chains and software interrupts) is very simple:

```
Interrupt:     Node - ties together server-chains
               Pointer to data-area for interrupt
               Pointer to interrupt code (terminates RTS)
```

Finally, the process structure is not so simple:

```
Process:  Task control block
          Message port used by DOS
          Process values - file handles for default IO, etc.
```

Examples

Two examples are given of communication between tasks, and communication between processes.

Task example

This can be compiled under Lattice in the usual way, using something like the following. Assuming that LC: INCLUDE: and LIB: have been assigned somewhere sensible, use

```
LC:lc1  -iINCLUDE:  -oRAM:  mtask
LC:lc2  -omtask.o  RAM:mtask
LC:blink FROM LIB:c.o+mtask.o TO mtask LIB LIB:lc.lib+LIB:amiga.lib
```

In this example a main task creates a sub-task then sends it a message containing a reason-code and a text pointer. It has to be careful that its child task has woken up and created its message port before it tries to talk to it; this is handled as follows:

1. First main creates its own message port.

2. Then it creates child task, and waits for a message from it.

3. When child task wakes up, it creates a message port, then sends main a message indicating if all is well (code zero), or if failed for some reason (code 1). This message does not need a reply.

4. Main then sends child a "hello there" message. Child gets the message and replies it; main gets the reply.

5. Finally, main sends child a special message telling child to go away. Child cleans up by removing its message port, then replies to main and enters an endless wait state. Main then deletes child, cleans up its message port, and exits.

This example aims for simplicity rather than beauty, and can be criticised in a number of ways.

1. The messages have been set up as static structures in the code. This makes the example easier to read, but it means it won't necessarily run on Amiga upgrades. The correct thing to do would have been to AllocMem some MEMF_PUBLIC memory, and copy the message data into it.

2. Child does not bother to check if more than one message is waiting. This is okay in this case, as we know main won't send any more until child has replied. However, if there was more than one possible source of messages to child, this would be dangerous.

3. The code makes use of "goto". Our own feeling is that "goto" is perfectly okay **for handling errors in nested structures only**. You may disagree.

Process example

In this example, MAINPROC and CHILDPROC are compiled separately. MAINPROC can be compiled as usual:

```
LC:lc1  -iINCLUDE:  -oRAM:  mainproc
LC:lc2  -omainproc.o  RAM:mainproc
LC:blink FROM LIB:c.o+mainproc.o TO mainproc LIB LIB:lc.lib+LIB:amiga.lib
```

CHILDPROC is linked without the Lattice standard startup code c.o which handles the normal business of startup from CLI or Workbench - we don't need this as we are kicking off this process ourselves. To do this, you have to compile with the -v option to disable Lattice stack-checking:

```
LC:lc1  -iINCLUDE:  -oRAM:  childproc
LC:lc2  -ochildproc.o  -v  RAM:childproc
LC:blink FROM childproc.o TO childproc LIB LIB:lc.lib+LIB:amiga.lib
```

This example is written to be similar to the previous one, and uses the same message structure. However, because child is now a process it can use AmigaDOS - it uses this to open a CON: window, where it prints out messages sent to it.

Again, it is necessary to be a bit careful with synchronisation at the beginning. In this case, this is achieved by main sending child a "wake-up" message to child's DOS message-port and waiting for a reply; this is okay, as DOS won't be using its port at the time. Thereafter, child's own message-port is used, which is safer. Main finds this message-port by looking in the "user-data" area of the task control block, where child has put a pointer to its message-port; alternatively, it would have been possible to use a public port.

Appendix 1 - Guru Meditation Mysteries

Guru alerts are the mechanism by which the Amiga system software informs the user of serious problems. Alerts are an Exec function, which can also be invoked through Intuition. They come in two forms - recoverable alerts from which you can return to normal multi-tasking, and dead-end alerts which necessitate a system reset. In the latter case the system normally puts up a requestor first (eg 'Software Error - Task Held') to allow you to go round saving files etc before this happens.

There are two principal sources of guru meditations - 68000 processor traps, and system software errors. If a task gets a 68000 trap it doesn't know what to do with - ie if it hasn't set up its own trap handling and hasn't had its traps "taken over" by a monitor like Wack - then the system will give a guru meditation such as the following:

```
 ----------------------------------------------------------
| Software Failure.  Press left mouse button to continue. |
|           Guru Meditation #00000003.000027D2            |
 ----------------------------------------------------------
```

Here the number before the dot is the 68000 trap number. Possible candidates are as follows:

```
     2     Bus error (hardware)
     3     Address error (word access on odd byte boundary - frequent!
     4     Illegal instruction (you are probably out of control)
     5     Divide by zero
     6     CHK instruction
     7     TRAPV instruction
     8     Privilege violation (supervisor instruction from user mode)
     9     Trace
     A     Opcode 1010 emulation (out of control again)
     B     Opcode 1111 emulation (          "         )
  20-2F    TRAP instructions
```

The number after the dot is the address of the task control block for the task that went wrong - almost certainly your task! From this point, you can go in with ROMWack to investigate further, as described in detail later.

The second type of Guru number is generated by the system software, and has the following form:

```
                         00  00  0000  .  00000000
     Block               A   B   C        D
```

Byte A defines in what part of the system software the alert was generated, and also flags if the alert is a recoverable one or dead-end. The most significant bit flags a dead-end alert; otherwise the values are as follows:

1	Exec library	10	Audio device
2	Graphics library	11	Console device
3	Layers library	12	Game-port device
4	Intuition library	13	Keyboard device
5	Maths library	14	Trackdisk device
6	Clist library	15	Timer device
7	AmigaDOS library		
8	RAM handler library	20	CIA resource
9	Icons library	21	Disk resource
		22	Misc resource
		30	Bootstrap
		31	Workbench

Byte B indicates the general cause of the problem, as reflected in the text error message:

1 No memory
2 Unable to create library
3 Unable to open library
4 Unable to open device
5 Unable to open resource
6 Input/output error

Word C gives more detail - its meaning varies depending on the source of the error (as specified in byte A) and can be found in the commented version of h-file exec/alerts.h. An example is

```
 ----------------------------------------------------------
| Not enough memory.  Press left mouse button to continue. |
|           Guru Meditation #02010009.0007D6B8             |
 ----------------------------------------------------------
```

This indicates a recoverable error from the graphics library (byte A = 02), and that the general cause of the problem is 'No memory' (byte B = 01). Refering to exec/alerts.h tells us that specific error word C = 0009 means 'no memory for TmpRas' - ie the graphics library was trying to allocate some memory for temporary storage during text or area-fill operations (TmpRas), and found it had run out of memory. The address 7D6B8 is just below the system stack at 7E800 on an unexpanded (512K) Amiga, so obviously we are running low on memory.

The number after the dot has three possible interpretations for this form of guru. In most cases, it is the address of the control block for the malfunctioning task, as in the traps case discussed above. In cases relating to memory allocation, it is the memory address which went wrong - an example is the Exec guru 81000009 'free twice' which indicates an attempt to FreeMem() some memory already in the system free list. Finally, in cases where the system is REALLY confused, to the point of not being able to find things like system task lists, the number after the dot is the ASCII text string 'HELP'!

Appendix 2 - About Semaphores

The issue of "contention" arises whenever two or more tasks want to share something - such as a bit of hardware like a port, or just some memory like a linked list or some other significant structure. Generally speaking, if you use the system software properly, then it will look after this for you; however, there are the following three exceptions.

1. If you want to access a bit of hardware directly, you should first of all claim it from the system by opening the appropriate resource, or calling an appropriate routine such as OwnBlitter(); when you have finished with it, you should then give it back by closing the resource, or calling DisownBlitter(). Programs (and programmers) who break this rule are increasingly referred to as "brain dead".

2. If you need to access a system linked list for some reason, you should first of all disable multitasking using Forbid(), and later restore it using Permit(); if the list may be accessed on the interrupts, then you should use Disable() and Enable() instead. An example of a system list that you may want to access is Intuition's list of "gadgets" attached to a window structure - if you do this, make sure you Forbid() first, or you may get into real trouble when Intuition tries to access the same list, running as part of input.device's task schedule. (A better alternative in this case is to detach gadget sub-lists before you look at them, using RemoveGList() and AddGList().)

3. If you are writing an Amiga application with two or more collaborating tasks or processes, then you will almost certainly find yourself in a position where two or more of your tasks or processes want to access the same data structures - which should be allocated MEMF_PUBLIC - and where you therefore need to be careful about contention. (An example from our work at Ariadne is the Amiga terminal for the Compunet network, which has processes concerned with upload/download from the net and others concerned with editing; these need to access a common linked-list of text and graphics information.) In this case, Exec can help you, using a powerful mechanism known as "semaphores".

Like a lot of Exec, semaphores were really put in for the convenience of the rest of the Amiga system software, particularly to cope with various contention issues involving Intuition. However, there is no reason why they shouldn't be used by application software, and indeed we would recommend you to do so. Before version 1.2, Amiga system tasks usually had to handle contention using the Forbid/Permit mechanism discussed in (2) above; this was unsatisfactory because it hung up the entire machine waiting for just two tasks which wanted to share something, and because it could result in various "deadlocks". This was sorted out in 1.2 by introducing semaphores.

A semaphore is essentially a flag which can be associated with something you want to share between tasks, such as piece of hardware, a structure or a linked list. Before accessing the shared object, a task must first of all "claim" the semaphore; if the semaphore has already been claimed by another task, then a mechanism exists to go into a WAIT state, and not be woken up again by Exec until the semaphore in question is available. There are essentially two mechanisms for semaphoring in Exec 1.2, a fast simple mechanism based just on task signalling which will do fine in most cases, and an alternative mechanism based on messages and ports, which is slower and more complicated, but sometimes more powerful.

Signal Semaphores

A "signalSemaphore" is an Exec structure which can be used for most cases of semaphoring, when a task needs either to claim a particular semaphore immediately, or to go to sleep until the semaphore is available. In our example of a linked list that we want to be able to share between two tasks, we might decide to have a signalSemaphore associated with the entire list. An Exec routine InitSemaphore() exists to initialise such a structure, so we could set it up by

```
OurListSem = AllocMem( sizeof(struct signalSemaphore), MEMF_PUBLIC);
InitSemaphore(OurListSem);
```

Having done this, before either task tries to access our shared list, it should claim the semaphore by

```
    ObtainSemaphore(OurListSem);
```

This will return at once if the semaphore is available; if not the task will WAIT on a signal-bit until the semaphore is free, at which point it will claim it and return from ObtainSemaphore. If you don't want to wait, but want to do something else if the semaphore isn't free, you can call AttemptSemaphore() instead; this will claim the semaphore and return TRUE if it is available, or give up and return FALSE if it isn't. (DON'T use this for "busy waiting"!) When the task has finished with the shared data, it must release the semaphore; this is done by

```
    ReleaseSemaphore(OurListSem);
```

Note that calls to ObtainSemaphore() can be nested; if you call it while you already have the semaphore in question, it will return immediately having incremented an "obtain count"; other tasks won't get a look in until you have dropped the obtain count to zero, by calling ReleaseSemaphore() once for every call you made to ObtainSemaphore(). Note also that tasks waiting for a given semaphore are put into a queue, so that more than one task can be waiting for the same signalSemaphore.

Named Semaphores

In order for two or more tasks to "rendezvous" on a signalSemaphore - to use it to control access to something or other - they obviously must all know where it is. This is a similar problem to a number of tasks being able to access each others' message ports, and is answered in the same way. If several tasks are closely collaborating, then they will probably be linked to form a single load-module, in which case they will all know where the semaphores are anyway; alternatively, if one task or process starts another, it can send it a "startup" message containing important information such as the locations of semaphores. We can think of these as "private semaphores".

Alternatively, it is possible to have "public semaphores" very like "public ports". Public semaphores must be given a unique name; they can then be linked into an Exec list of public semaphores at a given priority position, using a routine called AddSemaphore(); this is called instead of setting up a private semaphore using InitSemaphore(). Another task can then search for the semaphore by name using a routine called FindSemaphore(). Note that before the semaphore is deallocated, it should be removed from Exec's semaphore list; this is done by a routine called RemSemaphore().

Lists of Semaphores

Sometimes it is desirable to link semaphores together, for reasons other than being able to find them in Exec's public semaphore list. An example might be to control the linked list of "frames" of text and graphics information mentioned in the Compunet example; in this case we might chose to have a "master semaphore" looking after the whole list, with "sub semaphores" associated with each individual frame, themselves linked together in their own semaphore list. With this setup, if we want to lock a single frame we can simply ObtainSemaphore() its semaphore; if we want several frames we can first obtain the master semaphore to lock the whole list, then ObtainSemaphore() the frames we want, then release the master lock. If we want ALL the frames in the list, then we obtain the master lock, then we obtain all the individual semaphores using a routine ObtainSemaphoreList(); when finished, we can release them all using another special routine, ReleaseSemaphoreList().

Note that the use of a master semaphore associated with the entire list is essential, since otherwise you can get deadlock problems - say if one task is trying to obtain semaphore A followed by semaphore B, while another task is trying to obtain B followed by A. Note also that the use of semaphore lists in this way is incompatible with having named semaphores linked into an Exec list as discussed above; this is because the semaphore structure contains only one field for linking semaphores together!

As a final example of this sort of thing, consider the Layers library. Here we have a linked list of Layer structures each associated with a particular RastPort or Intuition Window; this is controlled by a master LayerInfo structure associated with the entire BitMap or Intuition Screen. Handling contention properly in this case is very important, to allow a number of programs to share the screen, each with their own private window(s); this is now handled internally by using semaphores in much the way discussed above.

Message based Semaphores

The signalSemaphore mechanism is simple, fast and powerful; its only drawback is that while you are in a WAIT state caused by calling ObtainSemaphore, you can't simultaneously be looking out for anything else - such as a different semaphore coming free, or an IntuiMessage indicating that the user has clicked a "GIVE UP" gadget. If this is a problem, you will have to use a slower and more complicated mechanism, based on messages and ports.

Using this mechanism, the semaphore is a special sort of public or private message port, set up with a special option PA_IGNORE telling the system not to signal any tasks when a message arrives, and with a field SM_BIDS initialised to -1. A task can "bid" for this semaphore, by calling an Exec routine

```
Procure(port, message);
```

where "port" is the special semaphore message-port, and "message" is a standard Exec message structure, initialised to contain the address of a suitable reply port. If the semaphore is available, Procure() will return TRUE immediately and the message will not be replied, and can be reused as soon as you feel like. Otherwise, Procure() will return FALSE, and the message will be replied as soon as the semaphore is available; you can therefore call Wait() with an approprate bit-mask to watch the designated reply port, while at the same time looking out for anything else of interest, such as other semaphore replies, Intuimessages, or whatever.

When you have finished with a message-based semaphore you should of course release it; this is done by another call

```
Vacate(port);
```

This will free the semaphore port for use by others, and cause the next task in line (if any) to be woken up, by getting a reply to its message.

```
/*************** Source file MULTITASK.C ***************

             Simple example of multi-tasking
                  Ariadne Software Ltd.
                       April 1986
*****************************************************/

/*** System header files required ***/

#include <exec/types.h>
#include <exec/ports.h>
#include <exec/tasks.h>


/*** Our own definitions ***/

#define CHILDSTACK 1000    /* stack size for child task */

struct OurMsg {   struct Message message;
                  LONG code;
                  char *text;
              };

#define OKAY       0  /* possible values for code field */
#define ERROR      1
#define CLEANUP    2
#define OTHER      3

/*** Exec Library routines used ***/

extern struct Message *WaitPort(),*GetMsg();
extern VOID PutMsg(),ReplyMsg();

/*** Exec Support Library routines used ***/

extern struct MsgPort *CreatePort();
extern struct Task *CreateTask();
extern VOID DeletePort(),DeleteTask();


/*** Variables accessible to both main & child tasks ***/

struct MsgPort *mainport;  /* pointer to main task's message port */
struct MsgPort *childport; /* pointer to child task's message port */
```

```
/****** Child task code ******/

childcode()
{
   struct OurMsg *childrcv;      /* pointer to message received by child */

   static struct OurMsg initmsg = {
      {  {NULL,NULL,NT_MESSAGE,0,NULL},    /* message.mn_Node */
         NULL,                             /* message.mn_ReplyPort */
         6                                 /* message.mn_Length */
      },
      0,                                   /* code */
      NULL                                 /* text */
   };

   childport = CreatePort(0,0);  /* create message port for child task */

   if (childport == 0) {
      initmsg.code = ERROR;
      initmsg.text = "Child task error";
      PutMsg(mainport,&initmsg); /* send message to main task */

      Wait(0);                   /* wait until child task deleted */

   } else {
      initmsg.code = OKAY;
      initmsg.text = "Child task okay";
      PutMsg(mainport,&initmsg); /* send message to main task */

      for (;;) {
         WaitPort(childport);    /* wait for message from main task */

         childrcv = (struct OurMsg *) GetMsg(childport); /* get message */

         if ((childrcv->code) == CLEANUP) {
            DeletePort(childport);  /* delete child task's message port */
            ReplyMsg(childrcv);     /* reply to main task */

            Wait(0);                /* wait until child task deleted */

         } else {

            /* process the message here */

            ReplyMsg(childrcv);     /* reply the message to main task */
         }
      }
   }
}
```

****** Main task code ******

```
main()
{
   struct Task *childtask;        /* pointer to child task's control block
   struct OurMsg *mainrcv;        /* pointer to message received by main */

   static struct OurMsg hellomsg = {
      {  {NULL,NULL,NT_MESSAGE,0,NULL},    /* message.mn_Node */
         NULL,                             /* message.mn_ReplyPort */
         6                                 /* message.mn_Length */
      },
      OTHER,                               /* code */
      "Hello child task"                   /* text */
   };

   static struct OurMsg finalmsg = {
      {  {NULL,NULL,NT_MESSAGE,0,NULL},    /* message.mn_Node */
         NULL,                             /* message.mn_ReplyPort */
         6                                 /* message.mn_Length */
      },
      CLEANUP,                             /* code */
      "Perform clean-up"                   /* text */
   };

   /* create message port & child task */

   mainport = CreatePort(0,0);    /* create message port for main task */

   if (mainport == 0) {
      printf("Failed to create main task's message port\n");
      goto error1;
   }

   childtask = CreateTask("child",0,childcode,CHILDSTACK);

   if (childtask == 0) {
      printf("Failed to create child task\n");
      goto error2;
   } else {
      printf("Child task created - waiting for message\n");
   }

   /* wait for initial messagre from child task */

   WaitPort(mainport);            /* wait for message to arrive */

   mainrcv = (struct OurMsg *) GetMsg(mainport);

   printf("Message received from child task:\n      %s\n",mainrcv->text);

   if ((mainrcv->code) == ERROR)
      goto error3;

   /* send a message to child task */

   printf("Sending message to child task:\n      %s\n",hellomsg.text);

   hellomsg.message.mn_ReplyPort = mainport;

   PutMsg(childport,&hellomsg);   /* send message to child task */

   WaitPort(mainport);            /* wait for reply */
   GetMsg(mainport);              /* get & ignore reply */

   printf("Reply received\n");
```

```
    /* send message telling child task to prepare for deletion */

    printf("Sending message to child task:\n      %s\n",finalmsg.text);

    finalmsg.message.mn_ReplyPort = mainport;

    PutMsg(childport,&finalmsg);  /* send message to child task */

    WaitPort(mainport);           /* wait for reply */
    GetMsg(mainport);             /* get & ignore reply */

    printf("Reply received\n");

    /* delete child task, clean up & exit */

error3:
    DeleteTask(childtask);        /* delete child task */

    printf("Child task deleted\n");

error2:
    DeletePort(mainport);         /* delete our message port */

error1:
    exit(0);
}

/********** End of Source file MULTITASK.C ************/
```

```
/*************** Source file MAINPROC.C ***************

        Main process for multi-processing example
                 Ariadne Software Ltd.
                      April 1986
*****************************************************/

/*** System header files required ***/

#include <exec/types.h>
#include <exec/ports.h>
#include <exec/tasks.h>
#include <libraries/dos.h>

#define TCBSIZE sizeof(struct Task)

/*** Our own definitions ***/

#define CHILDPRIORITY   0     /* priority for child process */
#define CHILDSTACK      4000  /* stack size for child process */

struct OurMsg {   struct Message message;
                  LONG code;
                  char *text;
              };


#define OKAY       0  /* possible values for code field */
#define ERROR      1
#define CLEANUP    2
#define OTHER      3


/*** Exec Library routines used ***/

extern struct Message *WaitPort(),*GetMsg();
extern VOID PutMsg();


/*** Exec Support Library routines used ***/

extern struct MsgPort *CreatePort();
extern VOID DeletePort(),DeleteTask();


/*** DOS Library routines used ***/

extern int LoadSeg();
extern struct MsgPort *CreateProc();
extern VOID Delay(),UnLoadSeg();
```

```
/****** Main process code ******/

main()
{
   struct MsgPort *mainport;  /* pointer to main process's reply port */
   struct MsgPort *dfltchildport; /* pointer to child process's default port */
   struct MsgPort *ourchildport; /* pointer to child process's new port */
   int childseg;              /* BCPL pointer to child's segment list */
   struct Task *childtask;    /* pointer to child's task control block */
   struct OurMsg *mainrcv;    /* pointer to message received by main */

   static struct OurMsg wakemsg = {
      {  {NULL,NULL,NT_MESSAGE,0,NULL},    /* message.mn_Node */
         NULL,                             /* message.mn_ReplyPort */
         6                                 /* message.mn_Length */
      },
      OKAY,                                /* code */
      "Wake up!"                           /* text */
   };

   static struct OurMsg hellomsg = {
      {  {NULL,NULL,NT_MESSAGE,0,NULL},    /* message.mn_Node */
         NULL,                             /* message.mn_ReplyPort */
         6                                 /* message.mn_Length */
      },
      OTHER,                               /* code */
      "Hello child process"                /* text */
   };

   static struct OurMsg finalmsg = {
      {  {NULL,NULL,NT_MESSAGE,0,NULL},    /* message.mn_Node */
         NULL,                             /* message.mn_ReplyPort */
         6                                 /* message.mn_Length */
      },
      CLEANUP,                             /* code */
      "Perform clean-up"                   /* text */
   };

   /* create message port */

   mainport = CreatePort(0,0);   /* create message port for main process */

   if (mainport == 0) {
      printf("Failed to create main task's message port\n");
      goto error1;
   }

   /* load & create child process */

   childseg = LoadSeg("childproc");

   if (childseg == 0) {
      printf("Failed to load childproc\n");
      goto error2;
   }

   dfltchildport = CreateProc("child",CHILDPRIORITY,childseg,CHILDSTACK);

   if (dfltchildport == 0) {
      printf("Failed to create child process\n");
      goto error3;
   } else {
      printf("Child process created\n");
   }

   childtask = (struct Task *) (((int) dfltchildport) - TCBSIZE);
```

```
/* send "wake-up" message to child process & wait for reply */

printf("Sending message to child process:\n      %s\n",wakemsg.text);

wakemsg.message.mn_ReplyPort = mainport;

PutMsg(dfltchildport,&wakemsg);     /* send message to child process */

WaitPort(mainport);                 /* wait for reply */

mainrcv = (struct OurMsg *) GetMsg(mainport);   /* get reply */

Delay(200);    /* delay for 4 seconds */

if ((mainrcv->code) == ERROR) {
   printf("Reply received - error in child process\n");
   goto error4;
}

printf("Reply received - child process okay\n");

/* find child process's newly-created message port */

ourchildport = (struct MsgPort *) childtask->tc_UserData;

/* send "hello" message to child process & wait for reply */

printf("Sending message to child process:\n      %s\n",hellomsg.text);

hellomsg.message.mn_ReplyPort = mainport;

PutMsg(ourchildport,&hellomsg);     /* send message to child process */

WaitPort(mainport);              /* wait for reply */

GetMsg(mainport);                /* get & ignore reply */

printf("Reply received\n");

Delay(200);    /* delay for 4 seconds */

/* send "clean-up" message to child process & wait for reply */

printf("Sending message to child process:\n      %s\n",finalmsg.text);

finalmsg.message.mn_ReplyPort = mainport;

PutMsg(ourchildport,&finalmsg);     /* send message to child process */

WaitPort(mainport);              /* wait for reply */

GetMsg(mainport);                /* get & ignore reply */

printf("Reply received\n");

Delay(200);    /* delay for 4 seconds */
```

```
   /* close down child process, clean up & exit */

error4:
   DeleteTask(childtask);     /* delete child process */

error3:
   UnLoadSeg(childseg);       /* unload the child process's code */

error2:
   DeletePort(mainport);      /* delete our message port */

error1:
   exit(0);
}

/********** End of Source file MAINPROC.C ************/
```

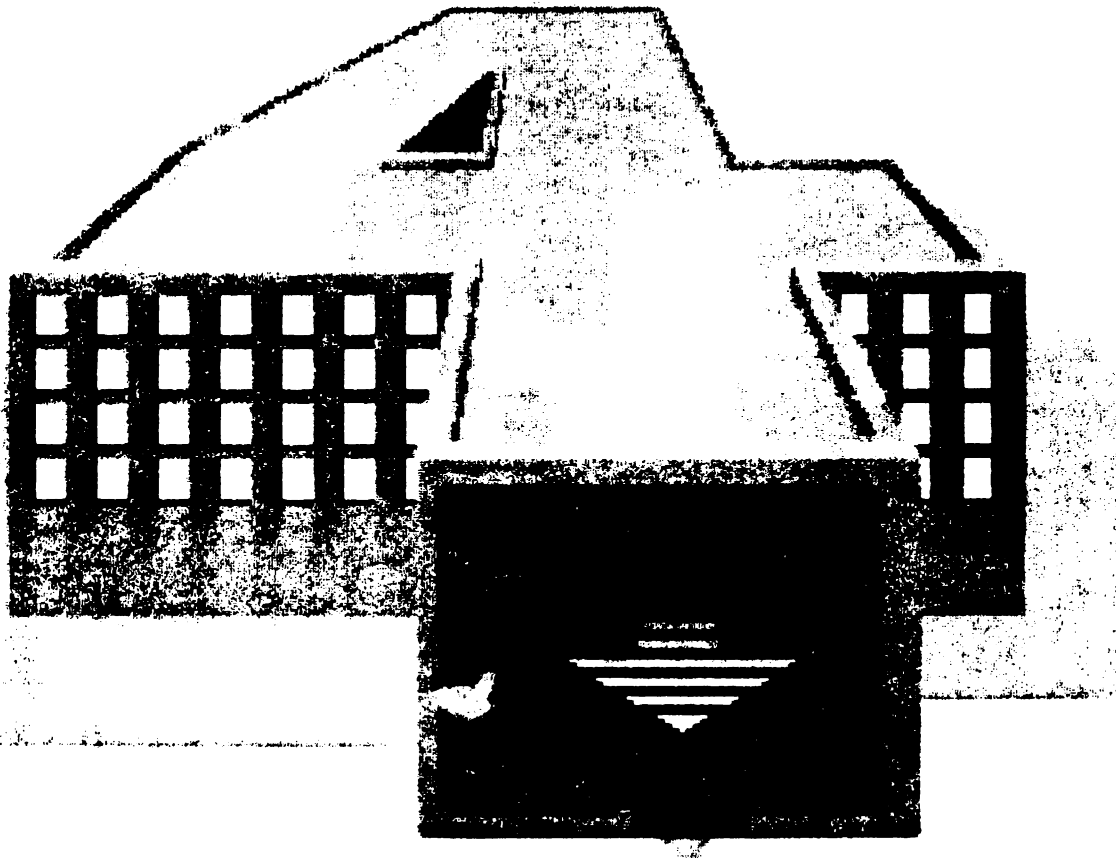
```
/*************** Source file CHILDPROC.C ***************

        Child process for multi-processing example
                  Ariadne Software Ltd.
                       April 1986
*******************************************************/

/*** System header files required ***/

#include <exec/types.h>
#include <exec/ports.h>
#include <exec/tasks.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>

/*** Our own definitions ***/

#define DOS_REV         29
#define INTUITION_REV   29

struct OurMsg {   struct Message message;
                  LONG code;
                  char *text;
              };

#define OKAY       0  /* possible values for code field */
#define ERROR      1
#define CLEANUP    2
#define OTHER      3


/*** Exec Library routines used ***/

extern ULONG OpenLibrary();
extern struct Task *FindTask();
extern struct Message *WaitPort(),*GetMsg();
extern VOID ReplyMsg(),CloseLibrary();


/*** Exec Support Library routines used ***/

extern struct MsgPort *CreatePort();
extern VOID DeletePort();


/*** DOS Library routines used ***/

extern int Open(),Write();
extern VOID Close();


/*** Declaration of function defined below */

extern VOID WriteString();


/*** Global pointers to required libraries */

ULONG SysBase;          /* base address for Exec Library */
ULONG DOSBase;          /* base address for DOS Library */
```

```
/****** Child process code ******/

main()
{
   struct Task *childtask;        /* pointer to child's task control block */
   struct Process *childproc;     /* pointer to child's process block */
   struct MsgPort *dfltchildport; /* pointer to child's default message port */
   struct MsgPort *ourchildport;  /* pointer to child process's new port */
   struct OurMsg *childrcv;       /* pointer to message received by child */
   int console;                   /* AmigaDOS file handle for console */
   ULONG *known;                  /* the only known address!! */

   known = (ULONG *) 4;
   SysBase = *known;              /* obtain Exec Library base address */

   /* wait for wake-up message from main process */

   childtask = FindTask(0);   /* find child's task control block */
   childproc = (struct Process *) childtask;
   dfltchildport = (struct MsgPort *) &(childproc->pr_MsgPort);

   WaitPort(dfltchildport);   /* wait for message from main process */

   childrcv = (struct OurMsg *) GetMsg(dfltchildport);   /* get message */

   /* create message port & open console for child process's output */

   ourchildport = CreatePort(0,0);  /* create message port for child process */

   if (ourchildport == 0) {
      childrcv->code = ERROR;     /* reply error message to main process */
      ReplyMsg(childrcv);
      Wait(0);                    /* then wait for deletion */
   }

   childtask->tc_UserData = (APTR) ourchildport;   /* point at port in TCB */

   DOSBase = OpenLibrary("dos.library",DOS_REV);

   if (DOSBase == 0) {
      DeletePort(ourchildport);   /* delete child's message port */
      childrcv->code = ERROR;     /* reply error message to main process */
      ReplyMsg(childrcv);
      Wait(0);                    /* then wait for deletion */
   }

   console = Open("CON:10/10/320/80/Child process",MODE_NEWFILE);

   if (console == 0) {
      CloseLibrary(DOSBase);         /* close DOS library */
      DeletePort(ourchildport);      /* delete child's message port */
      childrcv->code = ERROR;        /* reply error message to main process */
      ReplyMsg(childrcv);
      Wait(0);                       /* then wait for deletion */
   }

   WriteString(console,"Message received from main process:");
   WriteString(console,childrcv->text);

   /* tell main process we're okay */

   ReplyMsg(childrcv);
```

```
   /* wait for messages from main process & do appropriate action */

   for (;;) {
      WaitPort(ourchildport); /* wait for message from main process */

      childrcv = (struct OurMsg *) GetMsg(ourchildport); /* get message */

      WriteString(console,"Message received from main process:");
      WriteString(console,childrcv->text);

      if ((childrcv->code) == CLEANUP) {
         Close(console);                 /* close console for child process */
         CloseLibrary(DOSBase);          /* close DOS library */
         DeletePort(ourchildport);       /* delete child's message port */
         ReplyMsg(childrcv);             /* reply to main process */

         Wait(0);                        /* wait until child process deleted */

      } else {
         /* process other messages here */

         ReplyMsg(childrcv);  /* reply the message to main process */
      }
   }
}

/* Write a null-terminated string to specified file */

VOID WriteString(file,string)
int file;
char *string;
{
   int length;

   length = strlen(string);

   Write(file,string,length);
   Write(file,"\n",1);

   return;
}

/********** End of Source file CHILDPROC.C ************/
```

Amiga Libraries

How to Call a Routine, Without Knowing Where It Is



"System architecture" illustration by Hugh Riley.

Section 2 - Amiga Libraries

How to Call a Routine, Without Knowing Where It Is

When people leave behind machines like the 64 and come onto the Amiga, they usually start by asking questions like "Where is the screen RAM?" or "How do I poke the blitter?". Amiga initiates smile in a superior manner and gives answers like "That depends" or "You musn't". The beginner then asks for a memory map, and details of what addresses to call to invoke Intuition, Sound or Graphics; the expert smiles in an even more irritating fashion and explains that these questions too have no answer. It is the objective of this part of the Kickstart Guide to explain this.

This section is hard work and doesn't have many jokes in it. The whole question of how various aspects of the Amiga work together as a "soft machine" is not easy to understand or to explain; we found the only way to do it was to work through an example, looking carefully at what gets resolved where. This section therefore falls into two principal sections - a review of the concept of "libraries", and a detailed worked example. Note that the examples are based on USA Kickstart version 1.1 - I suppose we ought to have done them all again on 1.2, but we simply couldn't face it - besides which the principles involved haven't changed between versions.

PART I - PRINCIPLES OF LIBRARIES

The aspect of the Amiga which we are concerned with is known as soft machine architecture. This is a common feature of mainframes and mini-computers, and has been increasingly used in microcomputers since the first 16 bit machines. A "hard" machine architecture relies on absolute addresses, such as "To output a character you need to JSR $FFD2". "Soft" machine architecture means that the system relies on absolute addresses like this as little as possible. Instead, things are done in a more flexible way, in which system routines and jump tables don't always have the same addresses, but are instead put into memory wherever there happens to be room for them, as and when they are needed.

There are two main advantages to this. The first is that it leads to flexible and efficient memory utilisation, particularly in a multi-tasking system, and/or a system in which virtual memory techniques are likely to be used at some stage. The second is that it is very useful when it comes to upward compatibility - if there aren't any fixed addresses in the system, then you can produce an "upgrade" with everything in different places, and still quite legitimately claim compatibility. The main disadvantage is speed - if you can't rely on absolute addresses, then you have to keep using various forms of indirect addressing for everything, which is somewhat slower. It is because so much is done via DMA channels by the custom chips on the Amiga that you can get away with this.

Libraries, libraries and libraries

The mechanisms used to achieve a soft architecture on the Amiga are known as libraries. Unfortunately, this term is used in no less than three different ways with three different meanings! This is rather confusing, so it may be worth trying to clear up at the outset.

1. The Amiga ROM kernel uses "library" to mean a collection of routines in ROM or loaded off disk, accessed via a jump table attached to a "library" structure in RAM. The purpose of this is to allow languages or application programs to access different system functions in a controlled manner by suitable indirect addressing, discussed later in this section. Examples of libraries in this sense are the graphics library, the layers library, Intuition, DOS, etc. Very closely related to libraries are "devices" such as the parallel device, the serial device, the narrator and the clipboard, and "resources" such as "disk" and "cia". We shall refer to libraries in this context as "run-time" or "Exec" libraries.

2. The AmigaDOS linker uses "scanned libraries" to find standard definitions or routines used by the program being created; routines from the library are included in the output of the linker as necessary. Examples of scanned libraries are lc.lib containing all the lattice standard C-functions such as printf(), getc() etc, and amiga.lib, containing the Amiga functions. The latter fall into three categories - standard C functions, "kernel interface" functions (also known as "stubs") which provide the necessary intermediate code to invoke the run-time libraries, and "support" functions, which provide labour-saving routines to do things like creating message-ports. We shall refer to these as "scanned" or "linker" libraries.

3. The AmigaDOS technical reference manual also refers to "resident libraries" - however this does not appear to mean the same thing as the "run-time" libraries discussed above! In fact, in this context, the term "library" is a hang-on from Tripos, and can mean any loadable program module. In particular, "resident library" tends to be used to mean a set of routines which are linked separately, and then loaded and integrated with a controlling program at load-time rather than link-time; this can be a useful thing to do when developing long programs, as it reduces pressure on the linker. However, this is not something we have tried ourselves, and we will not be discussing it further.

(On the subject of sources of confusion, it may be worth mentioning the word "hunk", which is also used to mean three slightly different things in different contexts! We will tackle this later.)

Amiga run-time libraries

The concept of a jump-table will be familiar to anyone who knows the kernal on the Commodore 64 - it consists of a series of JMP instructions (6 bytes each on the Amiga) providing system routines with standard entry points which won't alter between versions. The Intuition library, for example, starts with a very large jump table, part of which may appear as follows:

```
00003D32        JMP  $FE0F9C   ;DrawImage
00003D38        JMP  $FE0F88   ;DrawBorder
00003D3E        JMP  $FE0F72   ;DoubleClick
00003D44        JMP  $FE0F66   ;DisplayBeep
00003D4A        JMP  $FE0F54   ;DisplayAlert
00003D50        JMP  $FE0F46   ;CurrentTime
```

There are two points to note about this. The first is that the JMP destinations are addresses in the ROM (or A1000 kickstart memory) - the addresses given are correct for the USA 1.1 kickstart, but won't be the same for the European version, or for 1.2. Provided the routines are accessed via the jump table this won't matter. The second point to note is that the jump table itself is in RAM; however, the crucial difference between this and the 64 is that these addresses are not constant - the system will have put the Intuition jump-table wherever it happened to find room for it, so you can't on future occasions just JSR $00003D44 and expect to get a display beep!

What does remain constant is the order of entries in the jump table. In fact, libaries are constructed so that the jump instructions build downwards in memory from a "library base address":

```
offsets

-114            JMP  $FE0F9C   ;DrawImage
-108            JMP  $FE0F88   ;DrawBorder
-102            JMP  $FE0F72   ;DoubleClick
-96             JMP  $FE0F66   ;DisplayBeep
-90             JMP  $FE0F54   ;DisplayAlert
-84             JMP  $FE0F46   ;CurrentTime

          (other jump table entries)

-24             JMP  $FD545A   ;EXTFUNC
-18             JMP  $FD545E   ;EXPUNGE
-12             JMP  $FD545E   ;CLOSE
-6              JMP  $FD5454   ;OPEN

0               ; library base address
```

Thus if we know the library base address, we can get the jump table entry we want by indirect addressing with a suitable displacement. For example, if we have the Intuition library base address in A6, then we can call DisplayBeep by

In assembler, you load A1 with a pointer to the name and D0 with the version number then invoke OpenLibrary as explained below; LibPtr is returned in D0.

OpenLibrary causes the Exec library manager to search its library list for a library with the name given, with version number greater than or equal to the one specified (so if you don't care, set Version = 0). If the library is found in memory, Exec calls a standard function OPEN in the library in question allowing it to take note of the fact that someone (else) wants to use it, then returns the library base address to the calling program, which then has all the information it needs to access the library. If the library is not currently in the library list, then Exec asks AmigaDOS to look for a file <LibName> in directory LIBS: (usually SYS:LIBS). If found this is scatter-loaded by AmigaDOS and linked into the library list; its OPEN routine is called and its base address returned. If both of these fail - i.e. the library asked for can't be found in memory or on disk - then Exec returns zero; it is up to the calling program to pick this up and do something about it, such as giving up.

This is all very well - you don't know where the libraries are but there's an Exec routine called OpenLibrary which tells you - but how do you call OpenLibrary if you don't know where it is? The answer is that Exec itself is organised as a library, so you call OpenLibrary via the Exec jump table, using a suitable offset from the Exec base address. To find the Exec base address, you look in the ONLY absolute software memory location on the Amiga (apart from the processor exception vectors), which is as follows:

```
 --------------------------------
|                                |
|                                |
|                                |
|                ---             |
| AbsExecBase  | 4 |             |
|                ---             |
|                                |
 --------------------------------
```

Complete Amiga memory map revealed!

(68000 experts will recognise that this is the same as the processor reset inital PC. However this is only needed during reset while there is a boot ROM switched in at the bottom of memory; after this it is free for use as AbsExecBase.)

Using system libraries

If you are using assembler, the process of using a library routine is as follows:

1. At start-up, you need to pick up the pointer to the Exec library base address at 4, and remember it as SysBase. This will be done for you if you link with a standard start-up module.

2. Before you use a library, you need to open it by

```
move.l    SysBase,A6
move.l    <name pointer>,A1
move.l    <version>,D0
jsr       _LVOOpenLibrary(A6)
move.l    D0,<library pointer>
```

3. To access a particular routine - say DisplayBeep in the Intuition library - you then proceed as follows

```
xref      _LVODisplayBeep
move.l    IntuitionBase,A6
jsr       _LVODisplayBeep(A6)
```

These operations can be performed by using a macro

```
LINKLIB   _LVODisplayBeep,IntuitionBase
```

which also pushes and pulls the previous contents of A6.

4. When you have finished with a library, you should close it, to allow, for example, memory occupied by a disk-loaded library to be reclaimed if it's wanted for something else. This is handled by an Exec routine CloseLibrary(LibPtr):

```
move.l    SysBase,A6
move.l    <library pointer>,A1
jsr       _LVOCloseLibrary(A6)
```

The offsets _LVOOpenLibrary and _LVOCloseLibrary can be declared external, or obtained by including exec_lib.i in the assembly.

From C, the process is very similar, except that you don't have to worry explicitly about the jump-table offsets, since these are got right by calling kernel interface functions (known as "stub functions") from the linker library.

1. In order to call Exec, you need to have a pointer SysBase obtained by reading location 4 - if you link with a standard startup like c.o, this will have been done for you, as will opening the DOS library and setting up a pointer DOSBase. (Note two errors in early versions of the ROM Kernel Manual on this score - the actual Exec library base address is known as SysBase not ExecBase, and the DOS

library base address is known as DOSBase not DosBase.)

2. In order to use Intuition, open the library by

```
IntuitionBase = (struct IntuitionBase *)
                    OpenLibrary("intuition.library",29);
```

   (Here we are using version number 29 which will open any Intuition revision from release 1.1 onwards - if you want to ensure you are running on 1.2, use version number 33.)

   IntuitionBase is a global variable declared

```
struct IntuitionBase *IntuitionBase;
```

   This form declares a pointer IntuitionBase to a structure also called IntuitionBase - if you don't like this, and aren't interested in the structure but just want to access the library, then you can use

```
APTR IntuitionBase;
```

   (When you link using amiga.lib, the linker will look for a global variable <u>of this name</u> from which to pick up the library base address, so you have to call it "IntuitionBase", and not "fred" or something.)

3. Having opened the library, you can then call intuition functions by name, eg

```
DisplayBeep(0);
```

   This causes a stub routine from amiga.lib to be included in the link, which sorts out registers as necessary (getting the zero into A0 in this case), picks up the library base address from IntuitionBase, then does the indirect call with the right offset.

4. When you have finished with Intuition, you close the library by

```
CloseLibrary(IntuitionBase);
```

The use of libraries from other languages varies in detail but tends to be similar in principle; for example in BASIC the command LIBRARY handles both opening the library and picking up information about function offsets and register usage from an appropriate file .bmap.

Register conventions

Amiga convention dictates that A6 should always be used to contain the base address of the library being called - note that you must ensure this, as the libraries may rely on it internally. D0, D1, A0 and A1 are scratch registers which are not preserved across library calls; these are also the principal registers used to pass values or pointers to the library routines, and to return results to the calling routine. Other registers should all be preserved across library calls.

(Some exceptions to this were discovered in version 1.1, in that some Intuition and graphics routines tended to destroy D6 and D7 - this could cause things that had been declared as register variables in Lattice C to be lost across library calls, which was a nuisance. Some interesting debate followed on the networks as to whether or not the library routines should in fact preserve all except the scratch registers, since pushing and pulling registers all the time can have a severe effect as regards performance. To our minds, this is an interesting question, but irrelevant to the main point, which is that if the documentation says the registers are preserved, then the registers should be preserved - this is fixed in version 1.2.)

Note also that this mechanism of communication using registers is designed for maximum efficiency when using assembler to call low level ROM kernel routines, also written in assembler. However, it is not particularly efficient for C, which passes values to functions by pushing them onto the stack before calling the function. This means that the amiga.lib link library stub functions have to read the appropriate values off the stack and put them into the right registers before invoking the run-time library; if the run-time library was itself written in C, the first thing it does is to push these registers back on the stack again, in order to set things up correctly to invoke a C function! While this is unfortunate, it is probably the best that could have been done given the requirement of a consistent interface to the kernel routines - it is obviously necessary that assembler calls to time-critical bottom-level routines should be as efficient as possible, even if this leads to some overhead when using higher-level aspects of the system. The only other alternative would have been to write the whole thing in assembler - this would have resulted in a typical assembler system, which is fast, efficient, streamlined, sexy, and not quite finished yet, sorry.

Library structure

The full structure of an Amiga run-time library is as follows:

```
          JMP  <routine n>      ;jump vectors
          .                     ;  - library-specific routines
          .

          JMP  <routine 4>
          JMP  <routine 3>
          JMP  <routine 2>
          JMP  <routine 1>

          JMP  EXTFUNC          ;reserved vectors
          JMP  EXPUNGE          ;  - standard routines for all libs
          JMP  CLOSE
          JMP  OPEN

Library:  Library node:  pointer to next lib in list
base                     pointer to previous lib in list
address                  node type 0
                         node priority 0
                         pointer to library name
          Flags byte
          Padding byte
          NegSize - size jump vectors in bytes (2 bytes)
          PosSize - size of data area in bytes (2 bytes)
          Library version number (2 bytes)
          Library revision number (2 bytes)
          Pointer to Id string, or zero
          Library checksum (4 bytes)
          OpenCnt - library open count (2 bytes)

          Data area follows
```

Meanings of the different library elements are as follows:

Jump vectors Used to access the library routines by suitable indirect addressing with negative displacement from the library base address, as explained above. See the ROM kernel manual volume 2 for full descriptions and register conventions for specific routines from specific libraries (V1.1), and/or see the 1.2 auto-docs. Jump vectors point either into ROM or kickstart memory (ROM libraries), or to wherever AmigaDOS happened to scatter-load the library routines (disk libraries).

Reserved vectors All libraries have to contain at least the following functions:

OPEN - Called when some task is going to OpenLibrary() this library. Should increment OpenCnt indicating that another task has this library open.

CLOSE - Called when some task has called CloseLibrary() for this library. Should decrement OpenCnt and do a "delayed expunge" (see below) if necessary.

EXPUNGE - Frees up memory allocated for this library, including the library node itself, and areas reserved by AmigaDOS for disk-loaded library functions.

EXTFUNC - Spare, reserved.

Note that after a library has been linked into Exec's library list, it will usually hang around in memory in case it is needed again, even if no task currently has it open. Libraries can be got rid of by calling RemLibrary() - when this happens the EXPUNGE routine is called for the library, allowing it to de-allocate resources such as memory. EXPUNGE checks OpenCnt; if this is zero, the expunge takes place immediately; otherwise a "delayed expunge" is flagged, and will take place as soon as OpenCnt becomes zero. Note that the Exec memory management routines will automatically RemLibrary all libraries with an OpenCnt of zero if they find they are running low of memory; thus a simple way of getting rid of all libraries that are not currently needed is to AllocMem more memory than can possibly be in the system!

<u>Library node</u> Links the library into Exec's library list. Also contains a pointer to the library name, used by OpenLibrary().

<u>Flags</u> Bit 3 of this byte is used to flag a "delayed expunge", explained above. Other bits are concerned with a checksum facility which exists to allow you to check library integrity - has some other task somewhere blown up and jumped all over this library?

bit 0	flags checksum in process
bit 1	flags one or more code vectors have been changed
bit 2	tells system to panic by issuing an alert (guru meditation) if checksum fails.

<u>NegSize</u> & <u>PosSize</u> The size in bytes of the jump table and data area respectively.

<u>Version</u> & <u>Revision</u> Version is the version number used by OpenLibrary(). Revision allows different revisions of the same version to be distinguished; this is not used by the system.

<u>IdString</u> Zero, or a pointer to an "id string" for this library - eg 1.1 Exec had id string "exec 31.34 (23 Nov 1985)".

<u>Sum</u> Library checksum.

<u>OpenCnt</u> How many tasks currently have this library open.

Data area A data area follows the Library structure; this can be used for work-space by the ROM libraries. It is particularly useful for variables that you might want to access from routines outside the library; this can be done using positive offsets from the library base address. For example, the Intuition data area starts with ViewLord - a View (graphics primitive) structure describing the current screen display - followed by a pointer to the currently active window, a pointer to the currently active screen, and a pointer to the start of a linked list of all current screens. Remember to Forbid() task-switching if you are going to access these lists directly!

Library facilities

Exec contains a number of functions documented in ROM kernel manual volume 1, allowing you to manipulate libraries. A summary of these is as follows:

AddLibrary(libPtr) - adds a new library to the system library list.

RemLibrary(libPtr) - removes library from the system library list, or flags a "delayed expunge".

MakeLibrary(parameter list) - a convenient way of constructing a new library. Handles creation of library node, calculation of checksum etc. Usually followed by a call to AddLibrary().

SetFunction(Library, FuncOffset, FuncEntry) - allows function with negative offset "FuncOffset" in library "Library" to be changed to point to "FuncEntry". Recalculates library checksum.

SumLibrary(library) - computes a new checksum and compares with the old one. If these values are different, and the library is not flagged as having been altered, then the system gives a guru alert; this is one way the system can realise that memory has been corrupted due to an "exploding" task.

System libraries

A list of libraries in the current system library list can be obtained by invoking OldWack then typing "libraries". These are as follows.

ROM libraries

The following libraries are in ROM (loaded from the Kickstart disk into protected memory on the A1000), and the appropriate library structures initialised in RAM during boot-up.

Exec - routines for linked list manipulation, task control, messages and ports, i/o handling, interrupt management, memory management, library, device and resource management. Also contains "processor control" functions to get condition codes, get/set processor status register, enter supervisor state or enter user state in a way which will work on processor upgrades (68010, 68020) - these are well worth using!

CLIST - This library was present in ROM version 1.1, but was dropped from 1.2, partly for lack of space, but also because no-one was using it. It contained Amiga string-handling, which worked using linked lists (like everything else on Amiga) - CList stands for "character list". Allowed you to initialise a block of memory for use as a "clist pool", then perform various string operations on clists within this pool, such as getting/putting bytes or words to start/end of list, converting clists to/from continuous data (such as null-terminated C strings), and performing concatenation, string-chopping, length and index operations. However, this library was roundly ignored by Lattice (and everyone else), on the grounds that they already had their own perfectly satisfactory string-handling routines, thank you.

GRAPHICS - Amiga graphics library, handling Views and Viewports (screen display primitives), RastPorts (drawing primitives), BitMaps (graphics data areas), GELS (graphics elements) consisting of SSprites (Simple Sprites), Vsprites (Virtual Sprites) and Bobs (blitter objects), Animobs (animation objects), text and fonts. Contains routines to control all the graphics facilities of the custom chips, including colour registers, the copper and the blitter, but allows you to take over for lower-level access if you ask nicely; "legal" access to these facilities should go by way of the graphics library.

LAYERS - Routines that work in conjunction with the graphics library, to allow a bitmap to be treated as a number of overlapping layers for window management. Handles manipulating a "damage list" of obscured regions for "dumb" refresh, buffering off obscured regions and rendering (drawing) into obscured regions for "smart" refresh, looking after "super-bitmaps", re-arranging and re-sizing layers, plus locking and unlocking layers as necessary to avoid contention problems.

INTUITION - Routines built on top of the graphics library, the layers library and the console device to provide a standard user-interface for different application programs. Handles screens, windows and borders, mouse and pointers, pull-down menus, gadgets, requestors, preferences, and I/O using the IDCMP (Intuition Direct Communications Message Port); also provides some convenience routines for "easy" memory management, graphics and text.

MATHFFP - Motorola fast (single precision) floating point routines for absolute value, testing against zero, comparison, addition subtraction multiplication and division, conversion to/from integer, negation.

DOS - AmigaDOS functions Open, Close, Read, Write, Protect etc - disk i/o and process control.

RAM-LIB - Ram handling for RAM-disk. Not accessible from C.

EXPANSION Routines invoked as part of the Amiga boot process, to handle external expansion on the A1000 or A500, or 'Amiga-side' expansion cards on the A2000. Handles interrogation of expansion cards to find out what they are, linking expansion memory into the Exec free-memory list, allocating memory etc to other expansion cards, giving them a change to initialise, etc.

Disk libraries

The following libraries may or may not be found if you use "libraries" in OldWack, since they live in LIBS: and are loaded by AmigaDOS as needed.

ICON - routines used by the Workbench to allocate/deallocate memory for Workbench objects (project, tool, drawer or whatever), to get/put Workbench objects and icons to/from .info files on disk, plus routines to deal with standard tool types, and updates to filenames ("copy of fred", "copy 2 of fred" etc).

MATHTRANS - routines for transcendental mathematical functions on fast floating point numbers - sin, cos, sin & cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, exp, ln, log, power, square root, plus routines to convert between fast floating point format, and IEEE standard double-precision floating point.

MATHIEEEDOUBBAS IEEE standard double precision basic floating point routines for same functions as MATHFFP routines. May be adapted to run with optional 68010/68020 floating-point co-processor in future.

TRANSLATOR Contains a single routine Translate() to convert English language to a phonetic string.

DISKFONT Contains two routines, to build an array of all fonts available in memory or on disk, and to load a font into memory from disk if necessary.

This completes a list of all the libraries admitted to in the Amiga documentation. Two further libraries can be found in the workbench SYS:LIBS directory, VERSION and INFO; the former contains a Workbench version number used by Workbench and DOS 'VERSION' commands and not a lot else, while the latter is used by Workbench to perform the INFO function from the "Workbench" menu.

Devices and resources

"Devices" and "resources" are two further Amiga concepts, both built on the library structure.

A "device" is a library with standard jump-table entry points used by Exec routines like DoIO(); these provide standard (physical device independent) IO functions such as reset, read, write, update and clear. Like libraries, devices can be resident or disk-loaded; standard resident devices are Timer, TrackDisk, Keyboard, Gameport, Input, Console and Audio; devices loaded from directory DEVS: (usually SYS:DEVS) are Narrator, Serial, Parallel, Printer and Clipboard. Devices in general, and the console device in particular, will be considered in detail in the next section.

A "resource" is a library WITHOUT the standard entry points - this is because resources are very closely tied to the Amiga hardware, and cannot be made to go away. Bits of hardware looked after by "resource" software are the four disk units, the two CIAs, the POTGO register (used to initiate a potentiometer read for joysticks etc), and "misc" - the serial and parallel port register bits. The function of the associated resource routines is to handle contention by granting or forbidding different tasks exclusive access to these bits of hardware. Usually this is handled for you by higher-level system software; however if you want to access this hardware directly, you can do it safely by opening the resource, then calling the appropriate routine - eg AllocUnit to allocate a disk unit. Note that the blitter is NOT considered a resource - though there was once discussion that perhaps it should be - but is looked after by routines like OwnBlit() from the Graphics library.

Linker scanned libraries

The Amiga linker scanned libraries consist of a concatenation of "hunks" each containing definitions of things like library offsets, or named routines which can be incorporated in the final disk-loaded program file if needed. The same file amiga.lib is used for both assembler and C programs, though a lot of the functions in it are only needed from C. Other languages may or may not use the same file.

As mentioned above, there are two principal scanned libraries used from C, lc.lib containing lattice standard functions, and amiga.lib containing the Amiga functions. Generally speaking, you need to link with both; if you are using Lattice 3.03 put lc.lib first if you are linking with LStartUp.obj, and amiga.lib first if you are using AStartUp.obj. Lattice 3.1 has a new Lattice startup c.o and doesn't support AStartUp.obj (shame!), so you have to put lc.lib first. In 3.1, the floating point maths functions have also been separated into different libraries; if you are using floating point maths, you have to put one of these before lc.lib. In this case, you have to decide which floating

point routines you want to use - standard Lattice maths functions from the linker library, fast floating point routines from ROM, or IEEEDoubBas from the LIBS: directory - then use the appropriate file lcm.lib, lcmffp.lib or lcmieee.lib. If you decide to use the fast floating point ROM routines, then you also have to tell the compiler to use FFP data format - this is done by setting the -f flag in LC1.

In addition to definitions like library offsets and current hardware memory locations, amiga.lib contains functions which can be divided into the following categories:

Kernel interface

"Stub" routines to sort out registers and call the specified run-time library routines; these routines expect the library in question to have been opened, and its base address put into an appropriate global variable GfxBase, IntuitionBase, DOSBase, etc. Interface routines are available for all run-time libraries except EXPANSION, RAMLIB, VERSION and INFO. Maths interface routines are supposed to handle conversion between C floating point number representation and Motorola fast floating point or IEEE double precision as necessary - this used to be tricky, but is got right by the new Lattice maths linker libraries.

Kernel support

There are two "linker only" sub-libraries within amiga.lib, providing support functions as follows:

EXEC SUPPORT Routines to handle initialising list headers, creating or deleting tasks, message ports, standard IO request blocks, extended IO request blocks. Ensures that this is done in a legal manner, and saves you the trouble of writing these functions yourself.

MATHLINK LIB Routines to convert fast floating point number representation to ASCII strings, "dual binary" format, or BCD; also a routine to round floating point strings. Note that the first of these was bugged and blew up on release 1.1 - it seems a bit dodgy on 1.2 as well, so it's probably safer to use the equivalent Lattice functions from lcmffp.lib.

Standard functions

Amiga.lib also contains the Amiga versions of some standard C functions - such as a limited (and much shorter than usual) version of printf() that makes use of Exec ROM functions. These will be used instead of the Lattice functions if amiga.lib is linked in front of lc.lib - but note that this doesn't work using Lattice 3.1, which is a pity.

Other linker libraries

There is an additional linker library file "debug.lib". This contains routines to handle formatting debug data according to C conventions - actually stub routines to undocumented entry points in Exec - and putting/getting debug information to/from a 9600 baud terminal attached to the Amiga serial port. Debugging techniques using the serial port will be looked at in detail later.

If you want to, you can also create your own linker libraries by concatenating object files. This is discussed in more detail below.

PART II - LINKING AND LOADING WITH AMIGADOS

To gain a further insight into how the C compiler (or other language), the linker, the relocating scatter-loader and the run-time libraries work together to produce a "soft machine", it is convenient to have a look at a real example, and consider what gets resolved where. An overview of this process is as follows.

1. For each module you compile (or assemble), the compiler (or assembler) converts your source code as far as possible into 68000 code, and resolves references to symbols defined within the same file (eg local variables), or whatever include files (.h or .i) you are using. The object file produced contains no absolute addresses; all references to internal symbols are stored as offsets. References to external symbols - ie symbols defined in other modules or in the linker libraries - are not resolved by the compiler or assembler; instead the name of the symbol to be referenced is output to the object file. Also output to the object file are the offsets corresponding to any symbols defined in this file that might be needed externally - for C these are the names of global variables and all functions not declared as "static".

2. The linker then joins together the object files for your different modules, together with whatever routines from the scanned libraries may be needed. This results in a "load file" containing no external references, but still with no absolute addresses - everything is given as offsets within different "hunks" in the load file.

3. AmigaDOS scatter-loads the load file; it first decides where in memory to put the various hunks, then loads them, converting hunk offsets to absolute addresses where necessary.

4. The program then is run from the start of the first hunk loaded - this will be the first module linked, which must therefore be some sort of startup module. As the code runs it will use various system libraries - it will open these and find out where they are as needed, as explained above.

Hunks, hunks and hunks

Before looking at this in more detail using a specific example, we need to sort out a couple of terms from AmigaDOS - "hunks" and "BPTRS". Like the word "library", the word "hunk" is used in three different contexts to mean slightly different things. These are as follows:

1. The output of the compiler is referred to as a "program unit". The program unit starts with a header giving the name of the unit - for Lattice 3.03 this is the same as the name of the .q file (quad file) which is the intermediate

file between LC1 and LC2 - eg "wombat.q" - while for 3.1 it is the name of the .o file (object file). Following this are three "hunks" containing code (CODE), initialised data (DATA) and uninitialised data (BSS or Base of Stack Segment - this need have nothing at all to do with stacks and is a silly name from Unix). "Hunks" in this context are known to Lattice as "segments", and described as 'P' (program), 'D' (data) and 'U' (uninitialised) - the size of each of these is given on termination by LC2. We shall call these hunks.

2. Each hunk can be divided into sub-units properly called "blocks", but sometimes also called hunks in order to confuse you. For example, a relocatable CODE hunk may start with a block of type "hunk_code" (confused yet?) containing the code itself, followed by blocks such as "hunk_reloc32" containing relocation information, followed by a block "hunk_ext" containing information about external symbols used by this hunk and global symbols defined within this hunk. Other blocks which may be present are "hunk_name" containing a hunk-name such as "text", "hunk_symbol" containing information about symbols which are to be passed onto the load file for use by a symbolic debugger like Wack or Metascope; and "hunk_debug" containing additional debug information like source-code line numbers. We shall refer to these as "blocks" - the idea of calling them "hunkettes" was considered, but regretfully rejected.

3. The output of the linker - the AmigaDOS "load" file - is also organised in hunks. These are very similar to "hunks" in sense 1, though with references to external symbols resolved. However, if two or more hunks have the same hunk_name, they are combined by the linker to form a "super-hunk" (our terminology), which is treated as a single unit by the loader; this is useful for things like named COMMON blocks in FORTRAN, or for data which is going to be accessed in certain ways like "base relative" addressing. This tends to happen to kernel-interface link library stub hunks, as all functions in the same library have the same hunk_name in amiga.lib. We shall refer to these as "super-hunks".

AmigaDOS BPTRs

A further bit of background needed to understand AmigaDOS is the data-type BPTR. AmigaDOS is written in BCPL; this is an ancestor of C which only supports one data type, which is a 4-byte address, or longword. A BPTR is a machine address expressed in longwords - i.e. it is the actual address divided by 4. This means that when moving between AmigaDOS and other aspects of the system, you have to keep shifting left or right by two to multiply or divide by four, in order to convert between BPTRs and ordinary machine addresses (APTRs) used by the rest of the system. This is a nuisance.

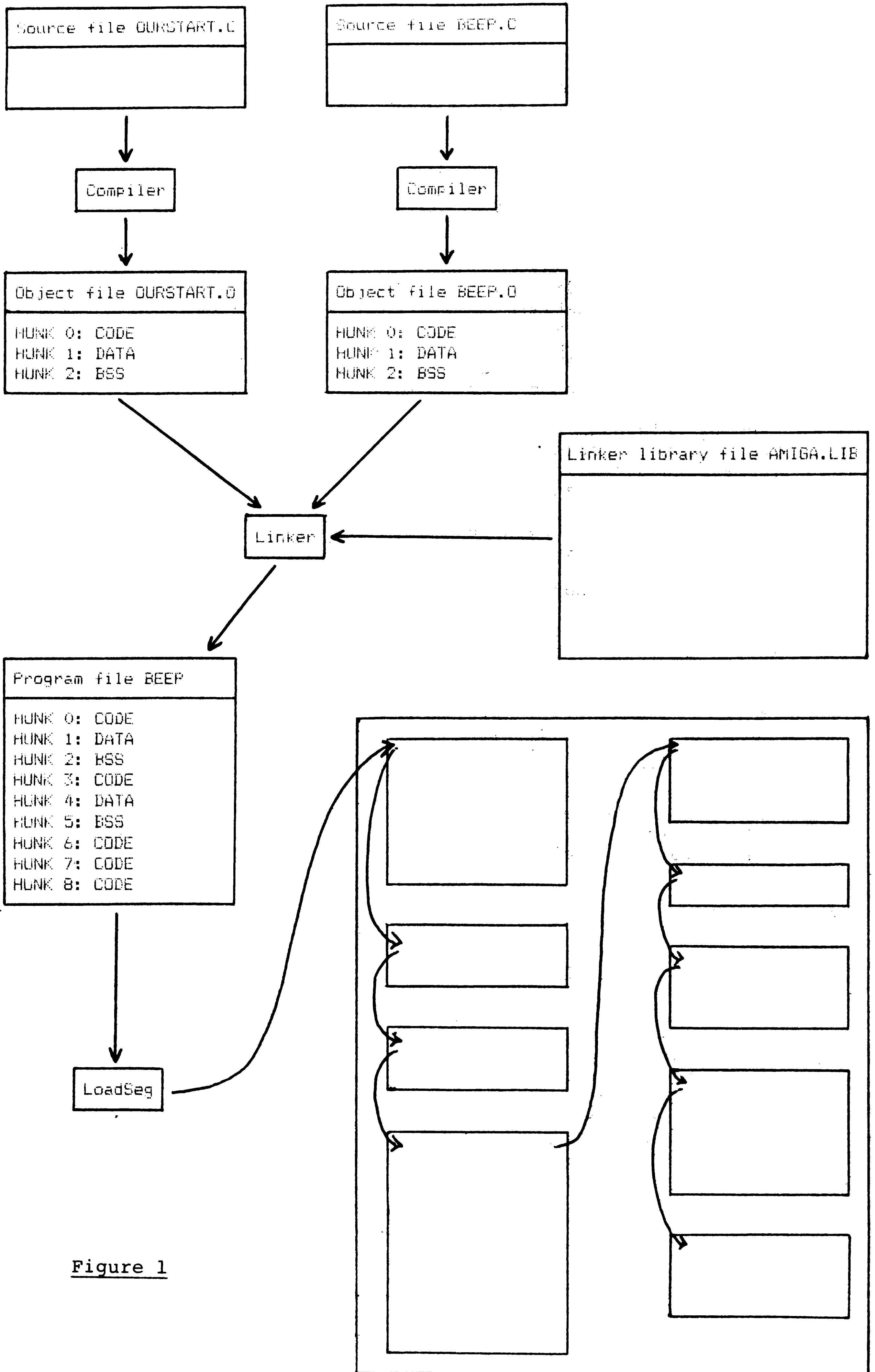


Figure 1

Now the example

In order to keep things as simple as possible, we will consider linking a minimal Intuition "screen beep" program (discussed in the "Getting started in C" section of this book) with a minimum startup module, providing the bare minimum needed to get a program started from the CLI.

The screen-beep program BEEP.C is listed at the top of figure 4; it attempts to open the Intuition run-time library, and if successful it beeps the screen 6 times, with a small delay between each obtained by calling the DOS Delay() function. It then closes Intuition and exits.

The minimum startup module OURSTART.C is listed at the top of figure 2. It picks up SysBase by looking in location 4, then opens the DOS library and sets up DOSBase - the example ignores the possibility that this might fail, though there's nothing it could do except give an alert if it did! It then calls main(), then closes the DOS library and exits.

(Of course, the standard startup modules do rather more than this, including remembering the entry stack level for use by the Exit() function, worrying about whether they were started from CLI or Workbench, and setting up vectors _stdin, _stdout and _stderr for use by the C standard functions, either by scanning a CLI command line or by waiting for messages from Workbench. Lattice startups like c.o do even more, including setting up locations used for stack-checking, and opening a window for stdin and stdout if necessary. Since we aren't using any C standard functions, aren't using Exit() and are only going to run from the CLI, we needn't bother with any of this.)

An overview of the process of compiling, linking then loading this example is shown in figure 1. Source files ourstart.c and beep.c are first of all compiled using LC1 and LC2, going through intermediate quad files ourstart.q and beep.q, and ending up with object files ourstart.o and beep.o. Each object file starts with a program unit header block, giving program unit names ourstart.q and beep.q (we were using Lattice 3.03), following by three hunks each containing code, initialised data, and uninitialised data (BSS). These hunks are identified by the type of one of the blocks in them, which is hunk_code, hunk_data, or hunk_bss, and numbered implicitly by the order they appear in the file. Note that Lattice always produces one hunk of each type, even if this results in "null hunks" having to be linked into the segment list at load time; however, these will be stripped out if you link using Blink.

Ourstart.o and beep.o are then linked, together with linker library file amiga.lib. This produces a program file beep consisting of a header followed by nine hunks - these are code, data and bss from main.o, code data and bss from beep.o, a code hunk from amiga.lib containing the _Delay stub code for the DOS run-time library, a "super hunk" containing the _CloseLibrary and

_OpenLibrary stub code to the Exec run-time library, and a hunk containing the _DisplayBeep stub code to the Intuition run-time library.

The program file is then loaded by AmigaDOS LoadSeg(), which can be invoked in various ways, the simplest of which is typing the program name "beep" at the CLI. This performs "scatter loading" by putting each hunk wherever there happens to be room in memory; as this is done any absolute memory references within the hunks are fixed up appropriately, as are all references from one hunk to another. The hunks are linked together in memory as an AmigaDOS segment list.

Finally the program is run, either as part of the process which invoked it by means of a JSR to the start of the segment list, as an AmigaDOS CLI "co-process" (see Section IV on AmigaDOS), or by being kicked off as a new process using AmigaDOS CreateProc().

Compiling the example

The process of compiling ourstart.c is illustrated in figure 2. The listing of ourstart.o given is based on the output of the Lattice Object Module Disassembler (OMD); this performs some integration of the blocks within each hunk to produce a more readable output.

Note that in this example, the DATA hunk contains just the null-terminated text-string "dos.library", while the BSS section contains room for two entries _SysBase and _DOSBase (the labels generated by the compiler are the same as the labels in the source code, with an underscore in front).

The CODE hunk generated by this example first loads A0 with 4, then moves whatever is pointed at by A0 to location 02.0000 - this means hunk 2 offset zero, which is where we are going to store _SysBase. It then pushes zero (library version number) and the address of hunk 1 offset zero (library name) to the stack - this is how Lattice passes values between functions - and calls an external routine _OpenLibrary. It then cleans up the stack, moves the contents of D0 to hunk 2 offset 4 (_DOSBase), then calls external routine _main. When this returns, it pushes hunk 2 offset 4 (_DOSBase) to the stack, calls external routine _CloseLibrary, cleans up the stack again, and exits.

A more detailed look at the start of ourstart.o is given in figure 3; this is obtained by annotating the output of the general purpose ObjDump utility, and shows details of exactly how something like "now I want to call a routine called _main but I don't know where it is yet" is represented. The file starts with a program unit header block - this starts off with a block identifier "hunk_unit", followed by the length of the name in longwords, followed by the name "ourstart.q" padded with nulls as necessary. This is followed by hunk 0, which contains CODE.

Source file OURSTART.C

```
#include <exec/types.h>
#define KNOWN_ADDRESS 4                    /* only fixed address in Amiga! */

extern APTR OpenLibrary();
extern VOID CloseLibrary();

APTR SysBase;                              /* pointer to EXEC library */
APTR DOSBase;                              /* pointer to DOS library */

OurStart()
{
   SysBase = *((APTR *) KNOWN_ADDRESS);    /* sort out sysbase */
   DOSBase = OpenLibrary("dos.library",0);    /* open DOS library */
                                              /* check for failure
                                                 & explode if necessary */
   main();                                 /* do whatever */
   CloseLibrary(DOSBase);
}
```

Compiler

Object file OURSTART.O

```
HUNK 0: CODE - length 0E longwords = 38 bytes

00.0000 _OurStart      MOVEA.L  #00000004,A0
00.0006                MOVE.L   (A0),02.0000
00.000C                CLR.L    -(A7)
00.000E                PEA      01.0000
00.0014                JSR      _OpenLibrary
00.001A                ADDQ.L   #8,A7
00.001C                MOVE.L   D0,02.0004
00.0022                JSR      _main
00.0028                MOVE.L   02.0004,-(A7)
00.002E                JSR      _CloseLibrary
00.0034                ADDQ.L   #4,A7
00.0036                RTS

HUNK 1: DATA - length 03 longwords = 0C bytes

01.0000                "dos.library"

HUNK 2: BSS  - length 02 longwords = 08 bytes

02.0000 _SysBase
02.0004 _DOSBase
```

Figure 2

```
Object file OURSTART.O

PROGRAM UNIT HEADER BLOCK

0000:            000003E7                      hunk_unit
0004:            00000003                      unit name is 3 longwords
0008:            6F757273746172742E710000      ourstart.q

HUNK 0: CODE

0014:            000003E9                      hunk_code
0018:            0000000E                      0E longwords of code
001C: 00.0000 207C00000004                     MOVEA.L  #00000004,A0
0022: 00.0006 23D0 00000000                    MOVE.L   (A0),02.0000
0028: 00.000C 42A7                             CLR.L    -(A7)
002A: 00.000E 4879 00000000                    PEA      01.0000
0030: 00.0014 4EB9 00000000                    JSR      _OpenLibrary
0036: 00.001A 508F                             ADDQ.L   #8,A7
0038: 00.001C 23C0 00000004                    MOVE.L   D0,02.0004
003E: 00.0022 4EB9 00000000                    JSR      _main
0044: 00.0028 2F39 00000004                    MOVE.L   02.0004,-(A7)
004A: 00.002E 4EB9 00000000                    JSR      _CloseLibrary
0050: 00.0034 588F                             ADDQ.L   #4,A7
0052: 00.0036 4E75                             RTS

0054:            000003EC                      hunk_reloc32
0058:            00000001 00000001             1 reference to hunk 1
0060:            00000010                      at offset 0010 in this hunk
0064:            00000003 00000002             3 references to hunk 2
006C:            0000002A 0000001E 00000008    at offsets 002A, 001E and 0008
0078:            00000000                      no more references

007C:            000003EF                      hunk_ext
0080:            81                            ext_ref32
0081:            000003                        3 longwords of symbol name
0084:            5F4F70656E4C696272617279      _OpenLibrary
0090:            00000001                      1 reference
0094:            00000016                      at offset 0016 in this hunk
0098:            81                            ext_ref32
0099:            000002                        2 longwords of symbol name
009C:            5F6D61696E000000              _main
00A4:            00000001                      1 reference
00A8:            00000024                      at offset 0024 in this hunk
00AC:            81                            ext_ref32
00AD:            000004                        4 longwords of symbol name
00B0:            5F436C6F73654C6962726172      _CloseLibrary
                 79000000
00C0:            00000001                      1 reference
00C4:            00000030                      at offset 0030 in this hunk
00C8:            01                            ext_def
00C9:            000003                        3 longwords of symbol name
00CC:            5F4F75725374617274000000      _OurStart
00D8:            00000000                      at offset 0000 in this hunk
00DC:            00000000                      no more externals

00E0:            000003F2                      hunk_end

00E4:            000003F2                      hunk_end
```

Figure 3

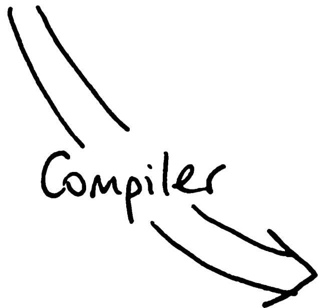
Hunk 0 starts with a block of type "hunk_code" containing 0E longwords of code; this is "partial code" containing zeros for unknown addresses such as the address of _OpenLibrary, and just offsets for locations in other hunks - see the hex corresponding to MOVE.L D0,02.0004 in fig 3. This is followed by a block of type "hunk_reloc32" containing relocation information for use by the scatter-loader; in this case, we tell the scatter loader that there is one reference to hunk 1 at offset 0010 in the code, and three references to hunk 2 at offsets 002A, 001E and 0008. By the time it processes this, AmigaDOS will have decided where to put hunks 1 and 2; the scatter loader will therefore be able to take the start addresses of these hunks and add them to the values already found at the offsets specified; this will convert the operand for MOVE.L D0,02.0004 to the correct absolute memory reference, by adding the offset stored in the code block to the hunk base address. Other relocation blocks hunk_reloc16 and hunk_reloc8 are handled in a similar manner.

Block "hunk_reloc32" is followed by a block "hunk_ext" giving information about external routines called from this hunk, and globals defined within the hunk. Within the overall block, there are three records of type "ext_ref32", indicating that there are three external symbols _OpenLibrary, _main and _CloseLibrary, used once each at offsets 0016, 0024 and 0030 respectively. These will be replaced by offsets within other hunks by the linker, and finally by absolute addresses by the loader. This is followed by one record of type "ext_def" indicating that there is one global symbol "_OurStart" at offset zero within the hunk, so if something else in the link wants to call _OurStart it knows where to find it. Generally, all function names will appear as global symbols in CODE hunks, unless they have been declared to be "static" to the compiler. Global variables will appear as "ext_def" in DATA or BSS hunks - examples (not shown in detail) are _SysBase and _DOSBase in hunk 2.

The process of compiling BEEP.C is illustrated in figure 4. 68000 fans should be able to follow the CODE hunk quite easily; note the use of LINK A6,-4 and UNLK A6 to grab some space off the stack for the automatic variable i.

```
Source file BEEP.C
```

```
#include <exec/types.h>

extern APTR OpenLibrary();
extern VOID CloseLibrary(),DisplayBeep();

APTR IntuitionBase;                         /* pointer to intuition library */

main()
{
   int i;

   IntuitionBase = OpenLibrary("intuition.library",29);  /* open intuition */

   if (IntuitionBase != 0) {                /* if open succeeded...    */

      for (i=0; i<6; i++) {                 /* six times...            */
         DisplayBeep(0);                    /* ... beep the screen...  */
         Delay(5);                          /* ... and pause           */
      }

      CloseLibrary(IntuitionBase);          /* and close again.        */
   }
}
```




```
Object file BEEP.O
```

```
HUNK 0: CODE - length 17 longwords = 5C bytes

00.0000 _main           LINK      A6,-4
00.0004                 MOVEQ     #1D,D0
00.0006                 MOVE.L    D0,-(A7)
00.0008                 PEA       01.0000
00.000E                 JSR       _OpenLibrary
00.0014                 ADDQ.L    #8,A7
00.0016                 MOVE.L    D0,02.0000
00.001C                 TST.L     D0
00.001E                 BEQ       00.0058
00.0020                 CLR.L     -4(A6)
00.0024                 CMPI.L    #00000006,-4(A6)
00.002C                 BGE       00.004A
00.002E                 CLR.L     -(A7)
00.0030                 JSR       _DisplayBeep
00.0036                 ADDQ.L    #4,A7
00.0038                 MOVEQ     #5,D0
00.003A                 MOVE.L    D0,-(A7)
00.003C                 JSR       _Delay
00.0042                 ADDQ.L    #4,A7
00.0044                 ADDQ.L    #1,-4(A6)
00.0046                 BRA       00.0024
00.004A                 MOVE.L    02.0000,-(A7)
00.0050                 JSR       _CloseLibrary
00.0056                 ADDQ.L    #4,A7
00.0058                 UNLK      A6
00.005A                 RTS
```

```
HUNK 1: DATA - length 05 longwords = 14 bytes

01.0000                 "intuition.library"
```

```
[illegible]: [illegible]    length 01 longword  = 04 bytes

[illegible] _IntuitionBase
```

Figure 4

```
Linker library file AMIGA.LIB

                                  .
                                  .
                                  .

HUNK WW: name "dos_lib" - CODE - length 05 longwords = 14 bytes

WW.0000 _Delay          MOVE.L   A6,-(A7)
WW.0002                 MOVE.L   8(A7),D1
WW.0006                 MOVE.L   _DOSBase,A6
WW.000C                 JSR      FF3A(A6)
WW.0010                 MOVE.L   (A7)+,A6
WW.0012                 RTS

                                  .
                                  .
                                  .

HUNK XX: name "exec_lib" - CODE - length 05 longwords = 14 bytes

XX.0000 _CloseLibrary   MOVE.L   A6,-(A7)
XX.0002                 MOVE.L   _SysBase,A6
XX.0008                 MOVE.L   8(A7),A0
XX.000C                 JSR      FE62(A6)
XX.0010                 MOVE.L   (A7)+,A6
XX.0012                 RTS

                                  .
                                  .
                                  .

HUNK YY: name "exec_lib" - CODE - length 06 longwords = 18 bytes

YY.0000 _OpenLibrary    MOVE.L   A6,-(A7)
YY.0002                 MOVE.L   _SysBase,A6
YY.0008                 MOVE.L   8(A7),A0
YY.000C                 MOVE.L   0C(A7),D0
YY.0010                 JSR      FDD8(A6)
YY.0014                 MOVE.L   (A7)+,A6
YY.0016                 RTS

                                  .
                                  .
                                  .

HUNK ZZ: name "intuition_lib" - CODE - length 05 longwords = 14 bytes

ZZ.0000 _DisplayBeep    MOVE.L   A6,-(A7)
ZZ.0002                 MOVE.L   _IntuitionBase,A6
ZZ.0008                 MOVE.L   8(A7),A0
ZZ.000C                 JSR      FFA0(A6)
ZZ.0010                 MOVE.L   (A7)+,A6
ZZ.0012                 RTS

                                  .
                                  .
                                  .
```

Figure 5

Linking

Relevant extracts from amiga.lib are shown in figure 5. Amiga.lib simply consists of a concatenation of program units; thus it is quite easy to make your own libraries by concatenating object files. Each hunk in amiga.lib is of CODE type, and starts with a block "hunk_name", which is the name of the associated library. The function names _Delay, _CloseLibrary are defined as external for use by the linker; they are also defined as symbols in an additional block "hunk_symbol", which causes their definitions to be included in the load-file, for use by Wack if necessary. The functions themselves are self explanatory; _DisplayBeep for example pushes the current value of A6 (must preserve regs except A0, A1, D0, D1), puts the externally defined _IntuitionBase into A6, reads the parameter from the top of the stack into A0, then performs the library call by JSR -96(A6). It then pops the previous value of A6 and exits.

The process of linking to produce a load file "beep" is illustrated in figure 6. Note that as two of the hunks from amiga.lib have the same name "exec_lib", these are concatenated to form one "super-hunk" by the linker. The linker also resolves all externals and replaces them by suitable hunk offsets. Figure 7 shows a "linker map" obtained by specifying "MAP beep.map" in the linker command string. This lists each hunk by type, memory type, and total size; it then gives file, program unit, base address and size, followed by a list of offsets for external symbols defined within the hunk. Note "super-hunk" number 7; the two hunks this comes from appear successively within the super-hunk, with a different base-address. The result of using the OMD on the load file is shown in figure 8; note the way that the externals have been resolved by the linker, but that there are still "hunk-relative" addresses to be resolved by the loader.

In addition to producing the output hunks illustrated, the linker also produces a "load file header" at the start of the file (not illustrated). This contains information about how many "resident libraries" (in the Tripos sense) to open, which will be zero under normal circumstances, plus the number of hunks in the file, and the length of each hunk in longwords. This is used by the loader.

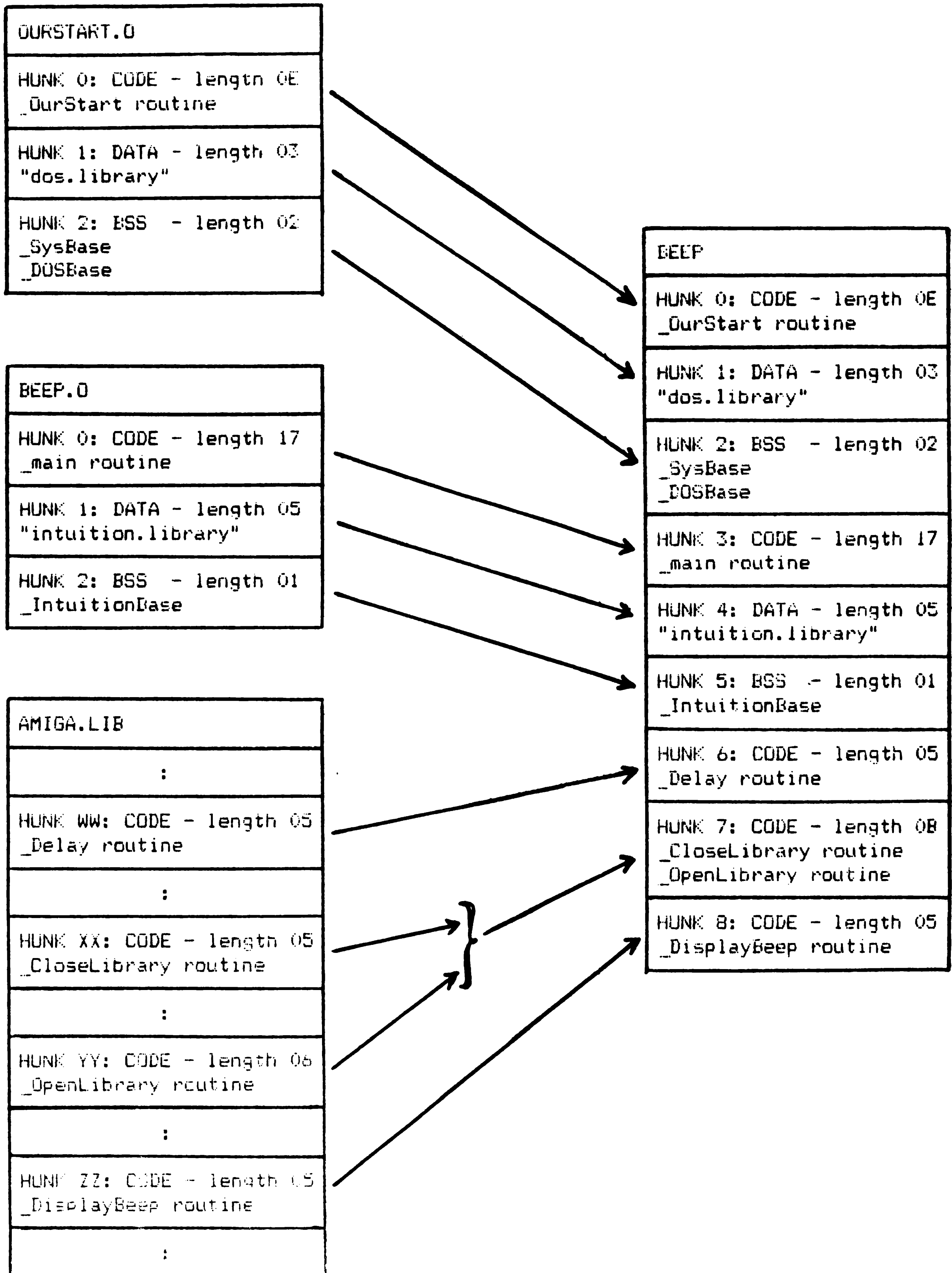


Figure 6

```
Linker map BEEP.MAP

0. CODE.    Memory Type PUBLIC Total Size 000038.

File: ourstart.o
Program Unit: ourstart.q
Base: 000000  Size: 000038

Symbol                             Value

_OurStart                          00000000


1. DATA.    Memory Type PUBLIC Total Size 00000C.

File: ourstart.o
Program Unit: ourstart.q
Base: 000000  Size: 00000C


2. BSS.     Memory Type PUBLIC Total Size 000008.

File: ourstart.o
Program Unit: ourstart.q
Base: 000000  Size: 000008

Symbol                             Value

_SysBase                           00000000
_DOSBase                           00000004


3. CODE.    Memory Type PUBLIC Total Size 00005C.

File: beep.o
Program Unit: beep.q
Base: 000000  Size: 00005C

Symbol                             Value

_main                              00000000


4. DATA.    Memory Type PUBLIC Total Size 000014.

File: beep.o
Program Unit: beep.q
Base: 000000  Size: 000014


5. BSS.     Memory Type PUBLIC Total Size 000004.

File: beep.o
Program Unit: beep.q
Base: 000000  Size: 000004

Symbol                             Value

_IntuitionBase                     00000000
```

Figure 7

```
6. CODE.   Memory Type PUBLIC Total Size 000014.
Hunkname: dos_lib

File: c1.1:lib/amiga.lib
Program Unit: No Name
Base: 000000  Size: 000014

Symbol                                 Value

_Delay                                 00000000


7. CODE.   Memory Type PUBLIC Total Size 00002C.
Hunkname: exec_lib

File: c1.1:lib/amiga.lib
Program Unit: No Name
Base: 000000  Size: 000014

Symbol                                 Value

_CloseLibrary                          00000000

File: c1.1:lib/amiga.lib
Program Unit: No Name
Base: 000014  Size: 000018

Symbol                                 Value


_OpenLibrary                           00000014


8. CODE.   Memory Type PUBLIC Total Size 000014.
Hunkname: intuition_lib

File: c1.1:lib/amiga.lib
Program Unit: No Name
Base: 000000  Size: 000014

Symbol                                 Value

_DisplayBeep                           00000000
```

Figure 7 (continued)

```
Program file BEEP

HUNK 0: CODE - length 0E longwords = 38 bytes

00.0000              MOVEA.L   #00000004,A0
00.0006              MOVE.L    (A0),02.0000
00.000C              CLR.L     -(A7)
00.000E              PEA       01.0000
00.0014              JSR       07.0014
00.001A              ADDQ.L    #8,A7
00.001C              MOVE.L    D0,02.0004
00.0022              JSR       03.0000
00.0028              MOVE.L    02.0004,-(A7)
00.002E              JSR       07.0000
00.0034              ADDQ.L    #4,A7
00.0036              RTS

HUNK 1: DATA - length 03 longwords = 0C bytes

01.0000              "dos.library"

HUNK 2: BSS  - length 02 longwords = 08 bytes

HUNK 3: CODE - length 17 longwords = 5C bytes

03.0000              LINK      A6,-4
03.0004              MOVEQ     #1D,D0
03.0006              MOVE.L    D0,-(A7)
03.0008              PEA       04.0000
03.000E              JSR       07.0014
03.0014              ADDQ.L    #8,A7
03.0016              MOVE.L    D0,05.0000
03.001C              TST.L     D0
03.001E              BEQ       03.0058
03.0020              CLR.L     -4(A6)
03.0024              CMPI.L    #00000006,-4(A6)
03.002C              BGE       03.004A
03.002E              CLR.L     -(A7)
03.0030              JSR       08.0000
03.0036              ADDQ.L    #4,A7
03.0038              MOVEQ     #05,D0
03.003A              MOVE.L    D0,-(A7)
03.003C              JSR       06.0000
03.0042              ADDQ.L    #4,A7
03.0044              ADDQ.L    #1,-4(A6)
03.0048              BRA       03.0024
03.004A              MOVE.L    05.0000,-(A7)
03.0050              JSR       07.0000
03.0056              ADDQ.L    #4,A7
03.0058              UNLK      A6
03.005A              RTS

HUNK 4: DATA - length 05 longwords = 14 bytes

04.0000              "intuition.library"

HUNK 5: BSS  - length 01 longword  = 04 bytes
```

Figure 8

```
HUNK 6: CODE - length 05 longwords = 14 bytes

06.0000                 MOVE.L   A6,-(A7)
06.0002                 MOVE.L   8(A7),D1
06.0006                 MOVE.L   02.0004,A6
06.000C                 JSR      FF3A(A6)
06.0010                 MOVE.L   (A7)+,A6
06.0012                 RTS

HUNK 7: CODE - length 0B longwords = 2C bytes

07.0000                 MOVE.L   A6,-(A7)
07.0002                 MOVE.L   02.0000,A6
07.0008                 MOVE.L   8(A7),A0
07.000C                 JSR      FE62(A6)
07.0010                 MOVE.L   (A7)+,A6
07.0012                 RTS

07.0014                 MOVE.L   A6,-(A7)
07.0016                 MOVE.L   02.0000,A6
07.001C                 MOVE.L   8(A7),A0
07.0020                 MOVE.L   0C(A7),D0
07.0024                 JSR      FDD8(A6)
07.0028                 MOVE.L   (A7)+,A6
07.002A                 RTS

HUNK 8: CODE - length 05 longwords = 14 bytes

08.0000                 MOVE.L   A6,-(A7)
08.0002                 MOVE.L   05.0000,A6
08.0008                 MOVE.L   8(A7),A0
08.000C                 JSR      FFA0(A6)
08.0010                 MOVE.L   (A7)+,A6
08.0012                 RTS
```

Figure 8 (continued)

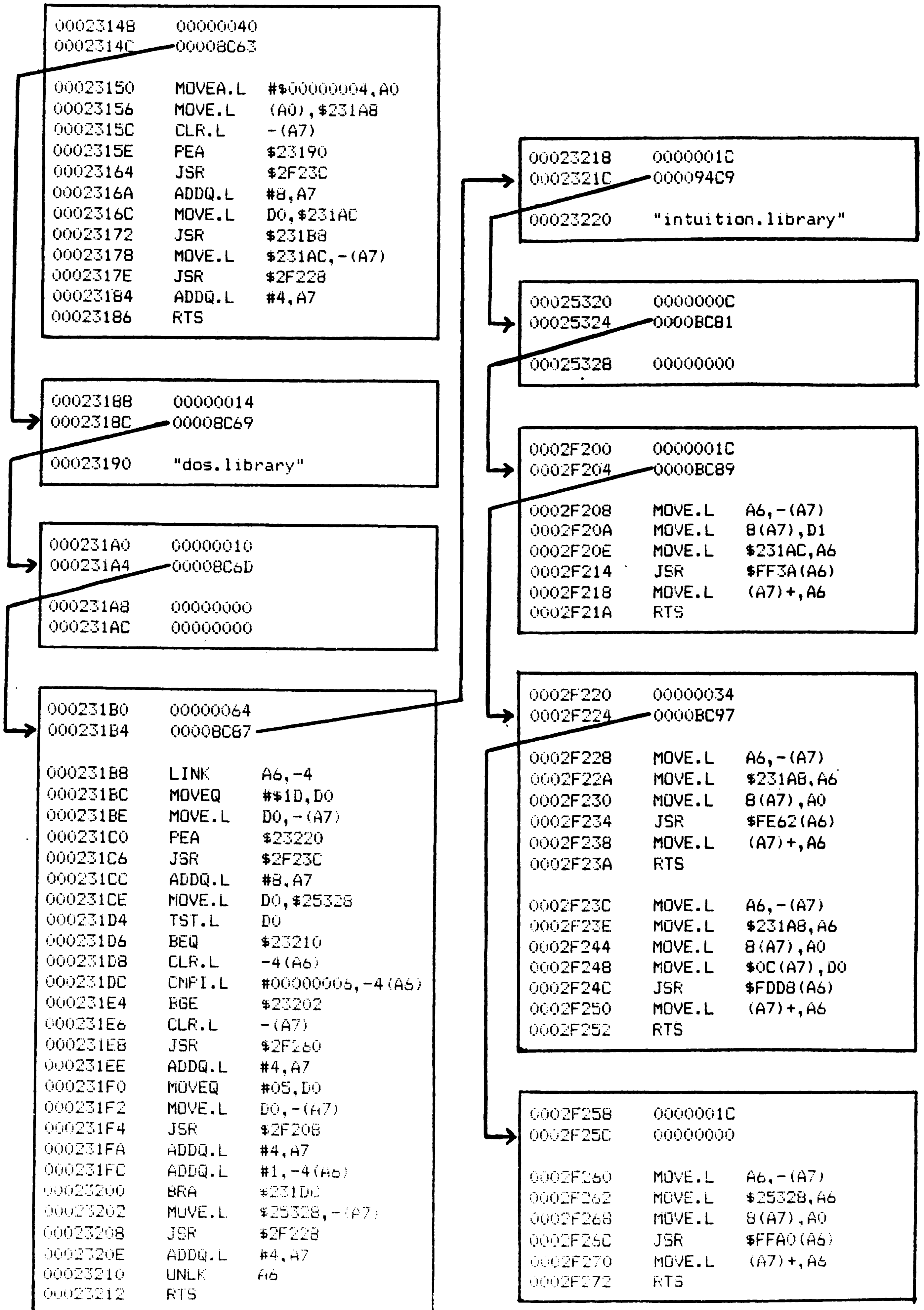


Figure 9

Loading

When the AmigaDOS scatter-loader is invoked by LoadSeg(), it first of all processes the load file header, by trying to allocate memory for each hunk, and if successful building up a "segment list" in memory, as illustrated in figure 9. Each record of this consists of a segment length in bytes, followed by a BPTR to the next segment (zero if last segment), followed by enough space for the hunk. Note that this allows AmigaDOS to know whether it has enough memory to be worth continuing with the load, and if so where in memory it is going to put each hunk, before it actually loads anything.

The load itself is now quite straightforward. The only things still needing attention in our example are the "hunk_reloc32" blocks, which specify where to modify the hunk being loaded by adding on appropriate hunk base addresses; the loader now knows what these are, so it can relocate as it loads, resulting in an arrangement in memory something like the one shown in figure 9. If we look at the code from the linker library that actually invokes DisplayBeep (7th segment), we see that the "hunk offset" in MOVE.L 02.0000,A6 has been replaced by the appropriate absolute address MOVE.L $231AC,A6; this sets up our Intuition library base pointer prior to the JSR -96(A6).

Running

When LoadSeg() has finished it returns a BPTR to the first segment in the segment list - in our example this will be $8C53, which multiplied by four gives $2314C. The code can then be run (usually as a CLI "co-process") by a call to this address plus 4, or set going as a separate process by calling CreateProc() - this is what happens when a program is run from Workbench.

The final piece of indirection involved in invoking the library is then resolved at run-time; in this case Exec returns $00003DA4 as IntuitionBase, which takes us to a jump vector at $00003D44, which takes us to a ROM routine at $FE0F66, which beeps the screen (at last) - see figure 10.

```
intuition.library

Offset   Description           Address    Value

FE5C     _LVOUnlockIBase       00003C00   JMP   $FE1284
FE62     _LVOLockIBase         00003C06   JMP   $FE1278
FE68     _LVOFreeRemember      00003C0C   JMP   $FE125E
                                  .
                                  .
                                  .
FF8E     _LVODrawImage         00003D32   JMP   $FE0F9C
FF94     _LVODrawBorder        00003D38   JMP   $FE0F88
FF9A     _LVODoubleClick       00003D3E   JMP   $FE0F72
FFA0     _LVODisplayBeep       00003D44   JMP   $FE0F66
FFA6     _LVODisplayAlert      00003D4A   JMP   $FE0F54
FFAC     _LVOCurrentTime       00003D50   JMP   $FE0F46
                                  .
                                  .
                                  .
FFDC     _LVOIntuition         00003D80   JMP   $FE0EEC
FFE2     _LVOOpenIntuition     00003D86   JMP   $FE0EE0

FFE8     LIB_EXTFUNC           00003D8C   JMP   $FD545A
FFEE     LIB_EXPUNGE           00003D92   JMP   $FD545E
FFF4     LIB_CLOSE             00003D98   JMP   $FD545E
FFFA     LIB_OPEN              00003D9E   JMP   $FD5454

0000     lib_Node.ln_Succ      00003DA4   00004E2C
0004     lib_Node.ln_Pred      00003DA8   00003A58
0008     lib_Node.ln_Type      00003DAC   00
0009     lib_Node.ln_Pri       00003DAD   00
000A     lib_Node.ln_Name      00003DAE   00FD5442

000E     lib_Flags             00003DB2   04
000F     lib_Pad               00003DB3   00
0010     lib_NegSize           00003DB4   01A4
0012     lib_PosSize           00003DB6   040E
0014     lib_Version           00003DB8   001F
0016     lib_Revision          00003DBA   0041
0018     lib_IdString          00003DBC   00000000
001C     lib_Sum               00003DC0   7DB20000
0020     lib_OpenCnt           00003DC4   0001
                                  .
                                  .
                                  .
```

Figure 10

```
wack beep
```

```
  Wack  Version 1.0

  loading symbols.........
  4 symbols loaded from 8 hunks


  generating symbol addresses ...
  ready

023150 207C 0000 0004 23D0      !......^D #..
go
seglist
#0   $02314C:00040 #1   $02318C:00014 #2   $0231A4:00010
#3   $0231B4:00064 #4   $02321C:0001C #5   $025324:0000C
#6   $02F204:0001C #7   $02F224:00034 #8   $02F25C:0001C

symbols
                         :
                         :
      base  00023150  00000000  ~hunk_0
      base  00023190  00000000  ~hunk_1
      base  000231A8  00000000  ~hunk_2
      base  000231B8  00000000  ~hunk_3
      base  00023220  00000000  ~hunk_4
      base  00025328  00000000  ~hunk_5
      base  0002F208  00000000  ~hunk_6
      base  0002F228  00000000  ~hunk_7
      base  0002F260  00000000  ~hunk_8
                         :
                         :
    offset  00000000  0002F228  _CloseLibrary
                         :
                         :
    offset  00000000  0002F208  _Delay
    offset  00000000  0002F260  _DisplayBeep
                         :
                         :
    offset  00000014  0002F228  _OpenLibrary
                         :
                         :

~hunk_0
023150 207C 0000 0004 23D0      !......^D #..
_OpenLibrary
02F23C 2F0E 2C79 0002 31A8   ?^N , y..^B 1..
where
_OpenLibrary + 0   (2F228 + 14)
quit
```

Figure 11

A final twiddle - ATOM

The above discussion represents the state-of-play as per Amiga version 1.0. However, it contains a shortcoming, in that there was no way in 1.0 of specifying what sort of memory each hunk was to be loaded into - chip memory (ie the bottom 512K accessible by the blitter etc), fast memory (up to 8M of expansion RAM not affected by the clever chips), or don't care. This was given a short-term fix in release 1.1, by means of a utility program called ATOM (Alink Temporary Object Modifier), used in conjunction with Lattice 3.03 and Alink.

When using Lattice 3.1 and Blink, ATOM is not needed, the same effect being obtained by new compiler switches, -c being used in LC2 to force code data or bss into chip memory (eg -ccdb would force all three), and -h (for high-speed) to force code data or bss into fast memory. It is very important to get this right if you want your code to run in Amigas with over 512K of memory; we suggest putting all data that needs to be in chip memory - like Intuition images, sprite definitions or audio waveforms - in a special source-module "slingthisinchip.c" or something, for compilation with the lc2 -c option.

The effect of running ATOM (after compiling or assembling, and before linking) or of using the new Lattice 3.1 compiler switches is to modify the .o files. This is done by modifying the type of one of the blocks in each hunk, which previously just marked the hunk as hunk_code, hunk_data, or hunk_bss. The two most significant bits of this longword are now used as follows:

0	0	Don't care - Use fast or chip, fast if available.
1	0	Use fast memory or fail
0	1	Use chip memory or fail
1	1	More info follows (reserved)

Having been modified, the .o files can now be passed to Alink (V1.1), or to Blink. This leaves these extra bits in the hunk types; it also ORs them into the most significant bits of the hunk lengths in the load file header, which are spare because the hunk lengths are given in longwords. Version 1.1 and 1.2 LoadSeg() then uses these bits to AllocMem() the right kind of memory for each hunk, or fails; old versions of LoadSeg() (1.0) will mistake hunks with non-zero values for these bits for very long hunks indeed, and fail with "out of memory".

A final point to note is that AmigaDOS refers to "don't care" memory as "public". This is confusing, as it does not appear to mean the same thing as "public" as in AllocMem(MEMF_PUBLIC), which is an upwards-compatibility feature, meaning memory that needs to be accessed by more than one task.

Last words on scatter loading

The clever way it handles relocating and scatter loading is probably the best thing about AmigaDOS. The fact that you can write fast position dependent absolute code, which will be fixed up to run properly anywhere by the loader, has a speed penalty while loading but leads to much faster code execution. In our opinion, this is much more sensible than other systems like OS9, which FORCE you to write position-independent code, with no JMPs, JSRs or other absolute memory references.

To take an extreme example, consider screen bit-plane access. Normally, you let the system allocate this for you - using a function like Intuition's OpenScreen() - then write into it using other functions in Intuition or the graphics library. However, if this isn't fast enough, and if you don't require system facilities like layers (windows) and so on, then you can find out where the memory has been allocated, then write into it yourself by appropriate indirect addressing - subject to various caveats discussion in Section 6. If even THIS isn't fast enough, then you can set up some bit-planes yourself by allocating some suitably enormous arrays compiled to be loaded as bss data into chip memory; you can then tell Intuition to use this for screen memory by passing OpenScreen() a NewScreen structure set up with a pointer to your own "CustomBitMap". This permits you to make ABSOLUTE references to your screen memory, which will be fixed up by the scatter loader. This proceedure isn't recommended; it is however as close as you can get to "poking screen RAM" on Amiga!

A final point - also relating to speed of execution - concerns two options available from Lattice 3.1, called base-relative and PC-relative addressing. Base-relative addressing is a form of data addressing in which one register (A6) is used to point at the start of a data area, and individual data items are accessed as 16-bit offsets within this area. This results in four bytes per data-reference (as opposed to six bytes for absolute addressing), and usually in faster execution; its disadvantage is that only up to 64K of data can be accessed like this, and that all this data must be in a continuous block. Base-relative addressing is specified by an option "-b" in LC1; this will cause the compiler to use base-relative addressing to access all data, and also cause the data and bss hunks to be given a special name "__MERGED", so that they will be joined together in a continuous super-hunk by the linker.

PC-relative addressing is a similar mechanism, this time applied to subroutine calls in code hunks; here the target address is specified as a 16-bit offset from the current PC-value, resulting in a four byte instruction rather than a six byte absolute JSR, which also executes faster. This is limited to offsets of plus or minus 32K from the current PC value; if you try to go further than this then BLINK will try to fix it up for you by generating a jump table, but this will obviously lose the code-length and speed advantage. It is therefore worth joining code hunks into super-hunks if you are going to use this; one way of doing this

is to use the -s option in LC2 which allows you to specify hunk_names, the defaults being "text", "data" and "udata". (The use of the word "text" to mean "code" is another silly name from Unix.) Note that using this option to join hunks into super-hunks also results in faster loading; its disadvantage is that it is more likely to fail in a system in which the available memory has become fragmented.

Try it yourself!

Of course, it is not necessary to understand very much of the above in order to do an Intuition screen beep! However, it is our feeling that an understanding of what happens "behind the scenes" is both interesting and useful, so we hope that you found the discussion above valuable, if exhausting.

Of course, if you really want an insight into the whole business, you should try your own example. If you want to do this, you will find the following utilities of interest.

1. ObjDump <filename> This can be used to examine ANYTHING made of hunks (object files, .lib files, load files), giving output in a "human" (Martian?) readable format, split into different hunks and blocks. Output can be redirected to the printer if desired.

2. OMD <filename> This is the Lattice Object Module Disassembler, which can produce a fairly readable disassembly of object files by integrating information in the various hunk blocks. OMD output includes labels defined in hunk_ext blocks in the file - if you want more symbolic information, then set the -d option in LC1 (version 3.1), which causes the compiler to output "hunk_symbol" blocks giving symbol definitions and "hunk_debug" blocks giving line numbers, which can be picked up by OMD.

3. Linker MAP and XREF options By specifying MAP <filename> and/or XREF <filename> in the linker command string, you can cause it to output a map similar to figure 7, showing each hunk in the output file, together with the symbols defined within it, or a cross-reference, which also shows where the symbols are referenced. This is useful in allowing you to spot symbols (and hence routines etc) which don't need to be public, or which don't get used at all!

4. Wack If you compile with the -d option in LC1, then hunk_symbol blocks are output by the compiler, and passed on by the linker into the final load file; these are usually then ignored by LoadSeg(). Note that amiga.lib contains hunk_symbol blocks for each library function, so these will always be present in the load file however you chose to compile; there is a Fish disk utility which allows you to strip these out of release software if you want to!
   If you want to make use of symbolic information while

debugging with Wack, then you can "bind" these symbols by invoking Wack by

```
wack <filename>
```

This should cause Wack to fetch the symbolic information while loading <filename>, and to calculate the actual address in memory corresponding to each symbol; note however that Lattice 3.1 introduces some new hunk types which old versions of Wack and Alink don't know about, so you may come to grief at this point with an error 'Unknown hunk type'. If the load does succeed, then you can get a list of symbols by typing

```
SYMBOLS
```

(Use right mouse button to stop them scrolling off the top of the screen.) This gives a list of internal symbols used by Wack - ie a list of Wack commands - plus the base address (actual start of code or whatever) of each hunk in the form ~HUNK_0 etc, plus the symbols picked up from hunk_symbol blocks _OpenLibrary etc. Other OldWack facilities of use in this context are as follows:

GO run program from current address - this will be initialised to the hunk 0 base address, so you can use GO to run the program just loaded, by a JSR to the start of the first segment.

SEGLIST gives address (real address not BPTR) of each segment of the program just loaded, and its length in bytes.

SYMBOLS gives all symbols as above. Typing in a symbol name (eg ~hunk_0) takes you immediately to that location.

WHERE gives your current position relative to the last symbol, else "You're lost".

LIBRARIES list of libraries in current Exec library list.

QUIT exit Wack.

See figure 11 for an example of using these facilities. Note that OldWack can't count - there are actually NINE hunks in our example, numbered 0 to 8!

5. Metascope If you can't get Wack to work - or if you just don't like Wack or can't get hold of a copy - then you can use an alternative symbolic debugger such as Metascope, which knows about the new Lattice hunk types. You can get Metascope to look at a particular program compiled with the -d option using

```
metascope <filename>
```

or alternatively you can load the program from within Metascope by selecting LOAD from the PROJECT menu. You can then get a list of symbols by selecting SYMBOL from the OPEN menu. You then can select particular symbols by pointing and clicking, open other windows with code-dumps or disassemblies at the locations corresponding to these symbols, plus all sorts of other wonders - see the Metascope documentation.

References

ROM kernel manual volume 2, Libraries. Contains much useful reference material (destined to become your most thumbed manual), plus a source-code example of a library.

V1.2 auto-docs for up-to-date library summaries.

AmigaDOS Manual Technical Reference - full information about binary file format, including various block types not covered above. Be warned that what this part of the DOS manual lacks in length it more than makes up for in total incomprehensibility, at least on a first reading. Be particularly wary of words which appear to mean the same thing as they do elsewhere in the documentation, but don't quite.

Amiga Devices

How to Perform IO, Without Worrying Too Much What To

"Into Amiga" illustration by Hanafi Houbart.

Section 3 - Amiga Devices

How to Perform IO, Without Worrying Too Much What To

A goal of many state-of-the-art machines, the Amiga (of course) included, is to provide "device independent IO". At first sight, this seems pretty daft, even by the standards of state-of-the-art buzz words - how can you address a tracker-ball exactly like a printer? The answer is that you can't, so the notion of device-independence should not be taken too literally; the more accurate alternative of "as device independent as possible under the circumstances" is however a bit long-winded.

In fact, the goal of device-independent IO is to provide as consistent an interface as possible to a wide variety of input and output devices. This is achieved by providing some standard structures and routines which are used in all IO, some standard input and output commands which are used by most IO, and a mechanism for adding the inevitable device-specific routines in a consistent and convenient manner. Besides conforming to abstract notions of "elegance", this has the concrete advantages of convenience to the programmer in not having to learn a whole set of new rules when considering a new IO device, and maximum ease when converting between devices which are in fact reasonably similar.

The software mechanisms used to handle IO on the Amiga are known generally as "devices"; examples of devices are the timer, trackdisk, keyboard, gameport, input, console and audio devices all of which are resident in ROM or "kickstart" protected memory, and narrator, serial, parallel, printer, and clipboard which are scatter-loaded off backing store as necessary.

Note that "devices" can appear at very different levels in the overall Amiga system architecture. The keyboard device for example is a very "low level" device which handles servicing keyboard interrupts, and passing "raw" key information to a higher level input coordinator called the input device. The console device on the other hand is a rather high-level part of the system with very close links to both Intuition and AmigaDOS; it takes keyboard information from the input device, and provides a variety of clever "virtual terminal" capabilities. Part 1 of this section considers devices in general; part 2 considers keyboard, input and console devices in detail.

Background concepts

In terms of software structure, a device is a special case of a general-purpose Amiga structure known as a library; "cunning" devices, capable of running quasi-independently of the calling program, have associated with them another structure known as a task. Roughly speaking, a library is a load of routines starting with a jump table, and a task is a mechanism used to allow different programs to share the CPU by "time slicing" on the interrupts - see the earlier sections of this guide for a full explanation of these concepts.

Devices and libraries

A device is a special case of a library, in that it consists of a jump table, followed by a node and various other library-type stuff, followed by a data area. As a library, it contains the standard entry points for Open(), Close(), Expunge(), and Extfunc(); in order to be a device, it must also have two further standard jump table entries BeginIO() and AbortIO(), which can be invoked by various standard routines in Exec.

Device:

```
    <other jump table entries>

    JMP ABORTIO     ;device standard entries
    JMP BEGINIO

    JMP EXTFUNC     ;library standard entries
    JMP EXPUNGE
    JMP CLOSE
    JMP OPEN

    Library node.

    Library flags, sizes, version, checksum and open count

    Data area follows.
```

The main general-purpose routine provided by all devices is BeginIO(); this is called with a pointer to a structure called an IORequest, which contains various information relevant to the IO call in question, including a "command" word specifying what is to be done - read data, write data, reset the device, or whatever. Commands fall into two categories. Standard commands such as read and write are satisfied by all devices if possible, though this may not be the case (you can't write to a tracker-ball!). Device-specific commands provide a mechanism to do things like allocating channels on the sound device, which wouldn't make a lot of sense to anything else.

In addition to device-specific commands, it is also possible for devices to have their own private additional jump-table entries beyond BeginIO() and AbortIO(), which provides an additional mechanism for device-specific functions, accessed by the usual way of calling functions from a library. Examples are the console device routines which provide translation from "raw" to "cooked" keyboard data by means of a key-table. In the case of the console, it is possible to open it just as a library, i.e. without linking it into the system's devices list or connecting it to the input device; it can then be used just for key translation (by Intuition or by the application program) without it doing anything else.

It can be seen that the mechanisms of standard commands, device-specific commands, and "private" jump-table entries provide considerable versatility in the way a device choses to function. Different devices make use of these mechanisms in different ways, some more neatly than others!

Devices and tasks

Besides the library structure discussed above, a device may have one or more tasks associated with it. This means that when the device is opened, one or more "task control blocks" may be linked into the task queues maintained by Exec, associated with routines within the device; this allows the device to run "alongside" other routines such as the application program, making use of Exec's ability to perform multi-tasking.

If this is the case, then each task will have associated with it a message port to allow queueing of IO requests. All IORequests start with a message structure, allowing them to be attached to a task's message port if appropriate. In this context, IO using BeginIO() (or higher level routines like DoIO()) can be viewed as a special case of the general Amiga mechanism of message passing. The IORequest structure can then be viewed as a message passed to the device; usually the device indicates that the IO request has been processed in the usual way, ie by replying the message to the calling program. (However, there is a shortcut to this process - see QuickIO below.)

To make this clearer, it may be worth distinguishing between "simple" devices with no associated tasks, and "cunning" devices which make use of multi-tasking; in this context, the keyboard device is (fairly) simple, while the input and console devices are both cunning.

A "simple" device performs IO in a way not very different from a Commodore 64. When an IO request is made, an IO request block is set up containing the relevant command - say to read a specified number of bytes of data. BeginIO() is then called with this request block and control passes to the device; control will not return to the calling program until the device has satisfied the IORequest and replied the IO request message with an error code

of zero, or else given up and returned an error. Thus the calling program has to wait until the data is available; this is known as synchronous IO, and is all that is available from a "simple" device.

A "cunning" device on the other hand has at least one associated task and message port, and is capable of asynchronous IO in which IO requests are queued to a message port, and other processing can continue while the device gets round to processing them. A cunning device maintains a "device busy" flag - when BeginIO() gets an IO request for something like "read data", it first checks if it is already busy, and if not gets on with the IO immediately. Otherwise, it queues the IO request to its associated task's message port. The associated task deals with removing IORequests and satisfying them as quickly as it can; the calling program will know when the request has been satisfied when the task replies the message. It can either hang around in a wait state waiting for this to happen (synchronous IO), or it can get on with something else in the meantime (asynchronous IO).

Note that you should never assume that something is a "simple" device, because someone may come along and re-write it. An IORequest should therefore always be viewed as a "message", ie as a bit of memory which is going to be "loaned" to another task - this means that it should be allocated MEMF_PUBLIC, and that it should NOT be modified by the calling program until the device has replied it. Note also that while a device may make use of multi-tasking to run asynchronously with the calling program, it will not attempt to perform more than one IORequest at once (!) - requests are queued to a message port, and dealt with one at a time. This is a nice simple idea sometimes known as single-threading - this is in order to sound better when you talk about it loudly in restaurants.

A good example of a "cunning" device with an associated task is the "input device" - this handles picking up "raw" input events from the keyboard, timer, and gameport all of which are simple devices, and passing them on to a "server chain" of input handlers, which includes Intuition. It is through the input device's associated task that Intuition "stays alive" while an application program is running; this is obviously rather important, which is reflected in the fact that the input device runs at the maximum priority used by the system, which is 20. (If you use Wack to look at this, you may see the "input.device" task more than once or not at all - this is because Wack is rather silly about tasks that may be moving between the Exec task-ready and task-waiting lists while Wack is looking at them!) The relationship of the various devices concerned with user input is discussed in detail in the next article.

(Since writing the above discussion of simple and cunning devices, it has been pointed out to me - hi Harry - that quite a few devices that I would have categorised as "simple", including keyboard.device and gameport.device, do in fact have interrupt handlers associated with them. While not being the same as the full mechanism of message-queuing used by a cunning device with an associated task, this does however give them a capability for elementary request queuing, and hence assynchronous IO. It may therefore be better to think of devices like this as only "fairly simple".)

Devices and Units

Associated with each device are one or more additional data structures known as units, arranged as follows:

Unit:

    message port for associated task (actual structure, not pointer)
    flags
    (padding)
    open count for this unit

This structure is initialised by the system, and a pointer to it returned as part of the process of opening a device; one of the parameters passed to the OpenDevice() function is a unit number.

In the case of the floppy disk drives, the unit number corresponds to the actual physical unit being accessed. Multiple drives are looked after by only one device structure and one set of code; however each unit has its own unit structure, its own message port, and its own TrackDisk.device task control block and data area for buffers.

In the case of other devices, the unit number may correspond to physical units (eg the gameports), it may be used for something else (eg for the timer it specifies whether to use vertical blank or an 8520 microhertz timer), it may have to be zero (eg the narrator), or it may be ignored completely (eg the audio device). However, in all cases, OpenDevice() always returns a pointer to a "unit" structure, as above.

Opening a device

Before using a device, it is necessary to open it, in the same way as opening a library. This is done by an Exec routine OpenDevice() which is passed a device name, a unit number, the address of an IORequest data structure, and some device-specific flags. Space must be reserved for the IORequest data structure, and its message header initialised before calling OpenDevice:

IORequest:

> Message structure - set up in advance with node-type NT_MESSAGE, priority zero, and appropriate pointer to reply port. Used internally by cunning devices to queue IO requests.
>
> Pointer to device base address - will be set up by OpenDevice().
>
> Pointer to unit structure - will be set up by OpenDevice().
>
> Command word - set up before calling BeginIO(). Not used by OpenDevice().
>
> Flags - set up before calling BeginIO(); not used by OpenDevice(). In theory, the lower four bits are for use by Exec - currently just to flag QuickIO - while the upper four bits are available for use as the device wishes. However, since the first thing that Exec SendIO() and DoIO() do is to blam absolute values into this location (versions 1.1 AND 1.2), this claim needs viewing with suspicion (unless you call BeginIO() directly).
>
> Error - error return, zero if successful.
>
> (Other stuff follows.)

The "other stuff" which follows depends on what sort of IO is to be performed. Some IO - eg the console - uses a structure called an IOStdReq, consisting of an IORequest structure followed by some more material as detailed below; a lot of other IO uses a extension to this consisting of an IOStdIO followed by some device-specific data. Exec support functions CreateStdIO() and CreateExtIO() exist to create the standard and the extended versions of these structures.

The full mechanism for opening devices and performing IO is summarised below:

1. Set up an IORequest structure followed by whatever else the device needs somewhere MEMF_PUBLIC, and initialise the message type and priority, and the address of your reply-port. If you are going to be using IOStdReq or extended IOStdReq structures, you can create these by calling

CreateStdIO() or CreateExtIO() respectively.

2. Open the device by calling

   OpenDevice(name, unit number, address of request
   structure, flags).

   This will attempt to open the device, scatter loading from disk if necessary, and call the device's own OPEN routine to allow it to initialise and connect itself in as necessary. It will also create a unit structure, and put the address of this structure and of the device itself into the IORequest structure. The use of unit number and flags varies from device to device - one use of flags is to request exclusive access, eg to the parallel device.

3. The IORequest structure can now be used in calls to BeginIO(), either directly or via other Exec calls (see below). The same structure can be used as was returned by OpenDevice, or copies of it can be made; before calling BeginIO() other fields should be set up, including the actual command to be performed. Command completion will normally be indicated by the IORequest message being returned to the reply port designated in stage (1).

4. Before your program exits, you should close the device using the Exec call CloseDevice().

In some other respects, devices behave just like libraries - eg it is possible to add new devices to the system lists or remove them using AddDevice() and RemDevice() respectively.

Standard commands

The standard commands supported by all devices (at least in as far as returning an error if they can't do them!) use a structure IOStdReq mentioned above:

IOStdReq:

    IORequest structure as above
    Number of bytes actually transferred (returned)
    Number of bytes we want transferred
    Pointer to data buffer
    Byte offset for structured devices (eg disk)

Note that only a few devices (eg the console) use this structure "as is"; most of them use an extended IO request block consisting of an IOStdReq followed by various device specific data. Examples of devices which do this are the clipboard, the narrator, parallel, serial and trackdisk devices; the trackdisk device for example uses a structure called IOExtTD, consisting of an IOStdReq followed by a disk change counter value, and a pointer used when accessing sector label information. Certain

devices ignore IOStdReq altogether, and just use an IORequest structure followed by the data in whatever form they feel like. Examples are the audio device, the printer, and the timer - the timer for example uses just an IORequest followed by a "timeval" structure, specifying time in seconds and microseconds.

The standard commands are as follows:

CMD_RESET - reset and reinitialise everything immediately, losing any pending commands.

CMD_READ - read bytes into data buffer.

CMD_WRITE - write bytes from data buffer.

CMD_UPDATE - ensure media up to date, eg no unwritten data lurking in internal disk buffers.

CMD_CLEAR - throw away contents of all internal buffers.

CMD_STOP - stop performing IO immediately - just queue requests.

CMD_START - start performing IO again.

CMD_FLUSH - immediately return all pending requests with an error.

"Cunning" devices maintain internal tables specifying which of these commands are for immediate execution, and which should be queued if necessary. Immediate commands are usually RESET, STOP START and FLUSH.

No frills IO - calling a device directly

The two standard routines supported by all devices are as follows:

BEGINIO - attempt to perform request specified
ABORTIO - attempt to abort request specified, by de-queuing it from a message-port.

The first of these routines can be accessed by an Exec support routine BeginIO(pointer to request structure), which handles picking up the device base address from the IORequest structure, setting up registers, then doing the appropriate indirect call to invoke BEGINIO. The second can be accessed via a routine AbortIO(structure) in Exec itself. (Note that the ROM kernel manuals are a bit confused about this - BeginIO is not mentioned in the Exec support documentation though it can be found in amiga.lib, and AbortIO is currently in Exec, not in Exec support as claimed!)

Thus the most direct way of performing IO is to set up your

request block with the command and other information you want (including the address of your reply port), then call BeginIO(). You can then go to sleep waiting for a reply (synchronous IO), or get on with something else checking for a reply from time to time (asynchronous).

QuickIO

The mechanism summarised above is general and powerful, particularly when it comes to cunning devices, internal request queuing, and asynchronous IO. However, in some cases - say when outputting characters one at a time to a "simple" device with no associated task - the mechanism of having to reply a message for every IO request adds a high overhead with no advantage, since the device won't be queueing requests internally anyway.

For this reason, if you don't require the full mechanism of asynchronous IO, it is possible to ask a device to "short-cut" if possible by setting a flag called QuickIO in the flags byte of the IO request block.

The QuickIO flag can be interpreted as telling a device that the calling program isn't particularly interested in getting a reply to its message. If a simple device gets an IO request with this flag set, it will perform the IO as usual, then return with this flag still set, and without bothering to reply the message. If a cunning device gets a QuickIO request, it may or may not be able to satisfy it. If it is not currently busy, it will perform the request immediately, and return with the QuickIO flag still set and without replying. However, if it is currently busy, it will have to queue the request to its message port. In this case it will return with the QuickIO flag clear, and later on when the request has been satisfied it will reply the message, to allow the calling program to re-use or deallocate the IO request block. Note that this means that you can't assume that QuickIO will be successful (even if you are talking to a simple device - someone might rewrite it!), and must always be prepared to cope with QuickIO failing, and the device replying your message. (If you find this too much of a pain, use the Exec routines discussed below, which handle this for you.)

Exec IO routines

In a few cases (eg when talking to the audio device) you have to use BeginIO() and AbortIO() directly - this is because these devices make use of flags which are jumped on by the routines from Exec. In other cases, it makes just as much sense to use the Exec routines, which are SendIO(), CheckIO(), WaitIO(), and DoIO(), all of which take a single parameter, which is the address of an IO request structure. The following details of the internal workings of these routines are based on version 1.1 (all right, we admit it, we disassembled the ROM!), but are unlikely to differ in any externally significant way in version 1.2.

SendIO(), CheckIO(), and WaitIO()

These routines are used for asynchronous IO - exactly what they do in version 1.1 is as follows:

SendIO() - Sends asynchronous IO request. Clears the QuickIO flag (by blamming zero in the IORequest flags byte), since you must have a message reply in order to do asynchronous IO. Then picks up the device base address from the IORequest, and calls BEGINIO for the device.

CheckIO() - Checks if asynchronous request has completed, and returns true or false accordingly. First checks for QuickIO flag still set (ie QuickIO was specified and succeeded), and if so returns true. Else checks if type of message has been changed to NT_REPLYMSG, and if so returns true; else returns false. In the later case, note that the reply will still need de-queueing from the reply port - this can be done by calling WaitIO().

WaitIO() - Waits for asynchronous IO request completion, and returns error code from IORequest block. Internally, works as follows (version 1.1):

First checks for QuickIO flag still set, and if so picks up the error code from the IO request block and exits.

Else picks up the reply port address from the IO request block, then picks up the corresponding signal bit number from the reply port.

Then clears the PAULA master interrupt enable (why?) and increments Exec's interrupt disable count.

Now looks back at the IO request block to see if its node-type has been changed to NT_REPLYMSG yet - if so the message has been replied, so it unlinks the reply from the reply port, decrements the interrupt disable nesting count and sets PAULA master interrupt enable if appropriate, picks up the error code and exits.

Otherwise it goes to sleep by calling Exec Wait(), waiting on the signal bit associated with the reply port. It then loops back to checking for NT_REPLYMSG, and exits or waits again accordingly.

DoIO()

This routine is used for simple synchronous IO, without you having to worry explicitly about messages and ports - its effect is to perform the IO, then return an error code.

Internally, DoIO first sets the QuickIO flag, since there's no particular reason for it to wait for a reply to its message. It then picks up the device base address, and calls BEGINIO; it then drops straight into WaitIO(), explained above.

References

ROM Kernel Manual Volume 1 contains a useful overview of IO, though it contains a few minor errors, and isn't very informative about QuickIO!

ROM Kernel Manual Volume 2 contains a detailed account of all the devices, including all the nasty non-standard bits. The example programs are particularly useful. The appendices contain a summary of all calls to all devices and listings of the .h and .i files giving the structures used by these calls; updates to these can be found on the 1.2 'docs' disk.

ROM Kernel Manual Volume 2 also contains an assembly-code listing for a "skeleton device", which is a "cunning" device with a task associated with it. This makes interesting reading.

Mouse & Keyboard

Ten different ways to get user input!

Elementary user input on the Amiga consists of the user pressing and releasing keys on the keyboard, moving the mouse and pressing and releasing mouse buttons. The system hardware detects these actions and system software deals with obtaining data from the peripheral chips and making these events known to higher-level system software and application programs in a convenient form.

The significance attached to these elementary events depends entirely on the context in which they happen. Thus pressing the left mouse button, moving the mouse and then releasing the left button may have any of the following effects:-

(1) requesting a disk copy (if the button was pressed with the pointer on a disk icon and released over another disk icon);

(2) resizing a window (if the button was used to select a sizing gadget;

(3) moving a window (if the button was used to select a window's drag bar;

(4) dragging an Intuition Screen up or down (if the button was used to select a screen's drag bar);

(5) selecting a window for input (if the pointer was inside a window when the button was pressed);

or a whole range of other possibilities.

Similarly, pressing and releasing the cursor-down key may have a number of different effects:-

(1) moving the cursor down a character line (eg when using Ed);

(2) moving the Intuition pointer down a raster line (simulated mouse movement - left Amiga key pressed);

(3) moving the Intuition pointer down by jumps (simulated fast mouse movement, left Amiga & SHIFT keys pressed);

(4) nothing (eg when inputting to a CLI window);

and so on.

The lowest level of system software handling user input deals with "raw" input events - that is key presses and releases etc - without attempting to attach any particular significance to them. Higher-level system software deals with such matters as generating an ASCII "a" code when the A key is pressed or an "A" code when the A key is pressed together with the SHIFT key, or interpreting a mouse button press as icon or gadget selection.

"Raw" key presses and releases are handled by the keyboard device, and mouse movement and button presses and releases by the gameport device (since the mouse is attached to one of the Amiga's game ports). These deal directly with the hardware and can be asked by other system software or by application programs to provide descriptions of "raw" input events in a standard format, called an "input event" structure. This contains data on what kind of event occurred, which key was pressed, etc, and a time stamp indicating when it happened, together with various other data. The definition of this structure can be found in the header file "devices/inputevent.h" for C programs, or in the include file "devices/inputevent.i" for assembler programs.

"Raw" keyboard events

The keys on the Amiga keyboard are numbered from 00 to 67 hex. The values attached to the keys have no relation to ASCII keycodes, they simply identify which key is being referred to. The keys can usefully be considered in three groups, classified by their raw keycode values:-

(1) keycodes 00 to 3F - correspond to ordinary printable characters, eg numerals, letters, punctuation characters;

(2) 40 to 5F - special keys, eg backspace, delete, function keys, HELP key;

(3) 60 to 67 - "qualifier" keys, ie. SHIFT keys, CAPS LOCK, CTRL key, ALT keys and "Amiga" keys, which generally don't mean anything on their own but can affect how other keys are interpreted.

When a key is pressed, the keyboard device generates an input event of class IECLASS_RAWKEY with appropriate keycode, eg the A key generates keycode 20 hex. When a key is released, the keyboard device produces an input event with the appropriate keycode but with bit 7 set, eg key A being released generates keycode A0 hex.

The input event also contains a description, as a set of bit flags, of which of the qualifier keys were down when the key was pressed or released, so that shifted A can easily be distinguished from unshifted A, etc.

Application programs may obtain these raw keyboard events directly from the keyboard device, as described in the ROM Kernel Manual, using CMD_READ commands (see Figure 1 (a)). This method is not recommended, since under most circumstances the input device (as described below) will be active, and it will also be requesting keyboard events, leading to a situation where some keyboard events are sent to the application and some to the input device, resulting in general confusion!

The keyboard device also handles the key combination ALT with both Amiga keys to produce a reset, and handler code to deal with clean-up processing before a reset occurs can be added to the system via the KBD_ADDRESETHANDLER command.

"Raw" mouse events

Mouse button presses and releases are treated by the gameport device in the same way as the keyboard device handles key presses and releases, where the buttons have "keycodes":

hex 68 - left mouse button
hex 69 - right mouse button
hex 6A - middle mouse button (if you have one)

except that the event class is IECLASS_RAWMOUSE instead of IECLASS_RAWKEY.

Mouse movements are also reported by using RAWMOUSE events, but with the value IECODE_NOBUTTON as the "keycode", indicating a mouse report not involving button press or release. Mouse position is reported for all mouse events, including button press & release.

The input device

Unprocessed raw input events are not the most convenient form for most application programs, so there are various levels of further system software provided to make life easier for the programmer. The key to these facilities is the input device. This is a task which requests raw input events from the keyboard and gameport devices, together with timer device events to handle key repeat timimg, etc and also receives notice of disk insertion and removal. It produces a single chain of input events, including handling key repeat by producing multiple key pressed events when keys are held down. Access to this chain of input events is not achieved by performing CMD_READ commands to the input device (as you might expect): instead these input events are passed to a chain of input event handling routines for further processing. An input event handling routine may do any of the following with the input event chain:

(1) handle the entire processing of an event and remove the event from the chain, so that further handlers are not aware that the event has occurred;

(2) remove an event from the chain, replacing it by another description of the same event, based on the context in which the event occurred;

(3) add new events to the chain;

(4) simply pass on an event unprocessed to other handlers (if any) in the chain.

When Intuition is active, it has an input event handler at high priority in the chain, and this is often the only handler present. Intuition's input event handler deals with such things as:-

(1) recognising mouse button events as system gadget selection and causing window sizing, dragging, pushing & popping, etc to happen, then "throwing away" the event;

(2) recognising mouse button events as window selection and re-directing input to the appropriate task;

(3) handling menu selection;

(4) recognising appropriate keyboard events as simulated mouse events, including menu short cuts;

(5) converting keyboard events to "cooked" form to produce ASCII character output.

Other input handling routines may be hooked into the chain by using the IND_ADDHANDLER command to the input device (as shown in Figure 1(b)). Since the priority of the handler is specified when it is added to the chain, handlers may be inserted before or after Intuition.

Key cookery and keymaps

For most purposes one is not interested in every key press and release, but in which ASCII characters and control sequences are being generated. The process of converting raw keyboard events into ASCII data we refer to as key cookery, since it produces "cooked" keyboard data as the result. The system needs some way of deciding what ASCII or extended character sequence to associate with a particular key when pressed with a given combination of "qualifier" keys such as SHIFT, CTRL and ALT. This is provided by means of a "keymap". You may provide your own custom keymap if you so desire, but there are default keymaps - for varous different countries - provided for use if you have no need of your own; these are installed by means of the 1.2 utility program SetMap. The keymap for raw keycodes 00 to 3F hex

(ordinary printable characters) is known as the Low Key Map, and that for raw keycodes from 40 hex upwards is known as the High Key Map. For each key with raw keycode in the range 00 to 5F hex (ie all except the "qualifier" keys), the keymap contains the following:-

(1) which of the "qualifier" keys SHIFT, ALT and CTRL (if any) affect the "cooked" keycode or sequence produced when the key is pressed - the possible "key types" are:

NOQUAL	no qualifiers - always generates the same result, regardless of qualifier keys;
SHIFT	the SHIFT key affects the output, but pressing ALT or CTRL doesn't make any difference;
ALT	the ALT key affects the output, but pressing SHIFT or CTRL makes no difference;
CTRL	the CTRL key affects the output, but SHIFT or ALT have no effect;
SHIFT+ALT	there are four possible results, produced by the key pressed alone, or with SHIFT, or with ALT, or with both SHIFT & ALT, but it makes no difference whether the CTRL key is pressed;
SHIFT+CTRL	there are four possible results, produced by the different combinations of the SHIFT and CTRL keys, with the ALT key making no difference;
ALT+CTRL	there are four possible results, produced by the different combinations of the ALT and CTRL keys, with the SHIFT key making no difference;
VANILLA	there are up to eight different results, produced by all possible combinations of the SHIFT, ALT and CTRL keys.

(2) for each combination of the qualifier keys, what single character or sequence is generated;

(3) whether the key is "capsable" - ie. does the key generate its shifted value when pressed with CAPS LOCK active;

(4) whether the key repeats when held down.

For a detailed description of keymaps, see the chapter on the Console Device in the ROM Kernel Manual.

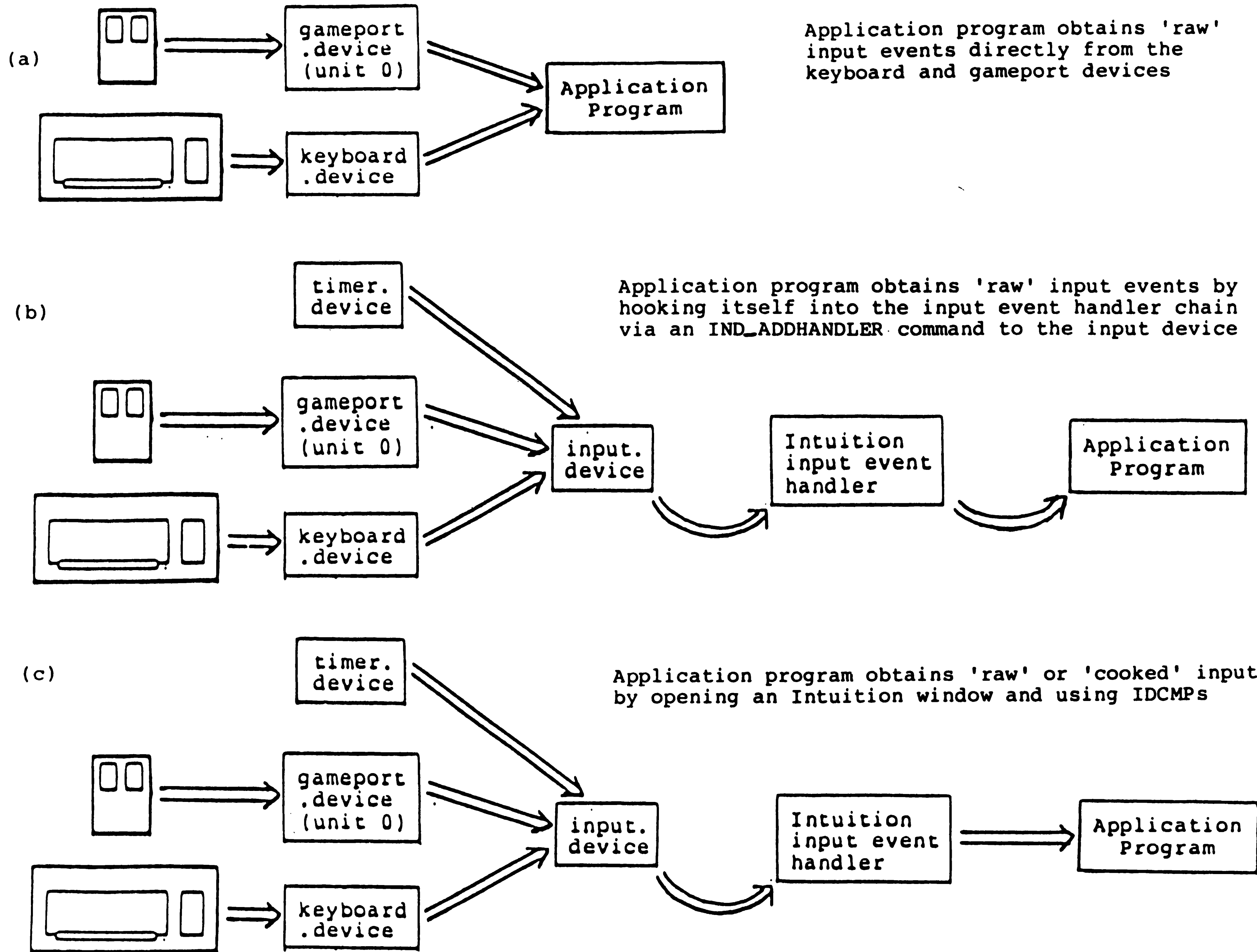
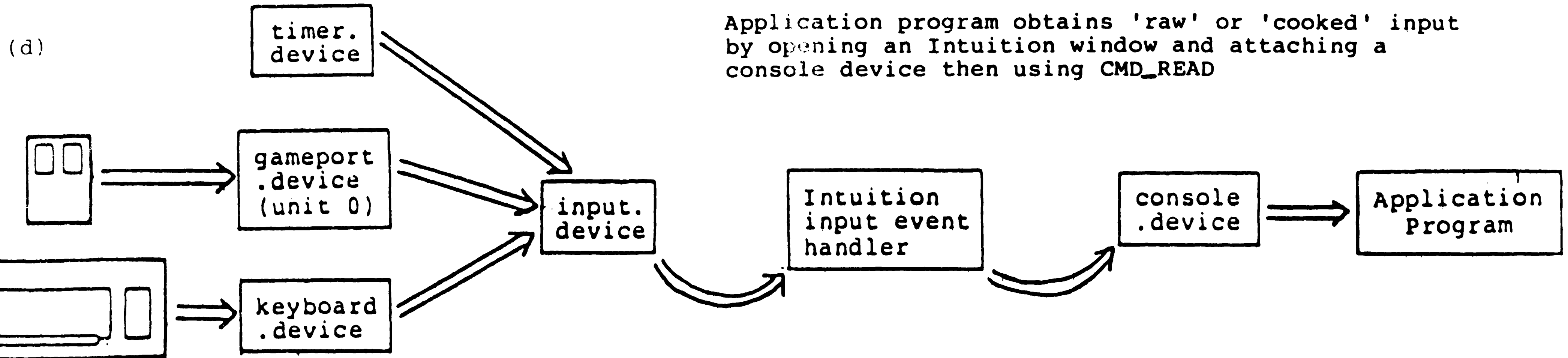


Figure 1

Figure 1 (contd.)

(d)



(e)

Application program obtains 'raw' or 'cooked' input
by opening an AmigaDOS 'RAW:' file & using Read()

timer.
device

gameport
.device
(unit 0)

keyboard
.device

input.
device

Intuition
input event
handler

console
.device

RAW:

Application
Program

The default keymap and CSI sequences

Table 1 contains a list of the keycodes generated by the "USA0" default keymap for all combinations of the three qualifier keys SHIFT, ALT and CTRL, together with the "key type" and whether the key is considered "capsable" or "repeatable". The following general points are of interest:-

(1) the only keys which don't repeat when held down are ENTER, RETURN, ESC and HELP;

(2) the letter keys are the only "capsable" keys;

(3) all keys in the Low Key Map (raw keycodes 00 to 3F hex) are of type VANILLA;

(4) the cursor keys, function keys and HELP key generate a sequence of characters, as does shift-TAB.

All of the multi-character sequences generated start with hex 9B, known as the Command Sequence Introducer, or CSI, and are of the form:

<CSI> [parameters separated by semi-colons] [space] <terminator>

where the optional parameters consist of ASCII digits or "?";

the space may be present or not, depending on the meaning of the sequence;

the terminator is a character in the range 40 to 7E hex.

Such CSI sequences are used both as the cooked form of special keys and also to send commands to the console device.

The console device

A convenient way for application programs to obtain input events after Intuition, and in a "cooked" form if required, without the programmer having to write his own input handling routines, is provided by the console device. This device consists of three principal elements:-

(1) an input event handler;

(2) conversion routines for doing key "cookery";

(3) output routines, to perform text output, scrolling, cursor movement, etc in a given window, normally using the standard character set.

The various ways of using these facilities are detailed below.

Table 1 - Default Key Map

Key Legend	"Raw" key codes		"Cooked" key codes											Key Legend
	key press	key release	key type	alone	with SHIFT	with ALT	with SHIFT & ALT	with CTRL	with SHIFT & CTRL	with ALT & CTRL	with SHIFT, ALT & CTRL	whether capsable	whether repeatable	
~	00	80	VANILLA	60	7E	E0	FE	00	1E	80	9E		REPEATABLE	` ~
1 !	01	81	VANILLA	31	21	B1	A1	31	21	B1	A1		REPEATABLE	1 !
2 @	02	82	VANILLA	32	40	B2	C0	00	00	80	80		REPEATABLE	2 @
3 #	03	83	VANILLA	33	23	B3	A3	33	23	B3	A3		REPEATABLE	3 #
4 $	04	84	VANILLA	34	24	B4	A4	34	24	B4	A4		REPEATABLE	4 $
5 %	05	85	VANILLA	35	25	B5	A5	35	25	B5	A5		REPEATABLE	5 %
6 ^	06	86	VANILLA	36	5E	B6	DE	1E	1E	9E	9E		REPEATABLE	6 ^
7 &	07	87	VANILLA	37	26	B7	A6	37	26	B7	A6		REPEATABLE	7 &
8 *	08	88	VANILLA	38	2A	B8	AA	38	2A	B8	AA		REPEATABLE	8 *
9 (	09	89	VANILLA	39	28	B9	A8	39	28	B9	A8		REPEATABLE	9 (
0)	0A	8A	VANILLA	30	29	B0	A9	30	29	B0	A9		REPEATABLE	0)
- _	0B	8B	VANILLA	2D	5F	AD	DF	1F	1F	9F	9F		REPEATABLE	- _
= +	0C	8C	VANILLA	3D	2B	BD	AB	3D	2B	BD	AB		REPEATABLE	= +
\ |	0D	8D	VANILLA	5C	7C	DC	FC	1C	1C	9C	9C		REPEATABLE	\ |
[undefined]	0E	8E												[undefined]
Num 0	0F	8F	VANILLA	30	30	B0	B0	30	30	B0	B0		REPEATABLE	Num 0
Q	10	90	VANILLA	71	51	F1	D1	11	11	91	91	CAPSABLE	REPEATABLE	Q
W	11	91	VANILLA	77	57	F7	D7	17	17	97	97	CAPSABLE	REPEATABLE	W
E	12	92	VANILLA	65	45	E5	C5	05	05	85	85	CAPSABLE	REPEATABLE	E
R	13	93	VANILLA	72	52	F2	D2	12	12	92	92	CAPSABLE	REPEATABLE	R
T	14	94	VANILLA	74	54	F4	D4	14	14	94	94	CAPSABLE	REPEATABLE	T
Y	15	95	VANILLA	79	59	F9	D9	19	19	99	99	CAPSABLE	REPEATABLE	Y
U	16	96	VANILLA	75	55	F5	D5	15	15	95	95	CAPSABLE	REPEATABLE	U
I	17	97	VANILLA	69	49	E9	C9	09	09	89	89	CAPSABLE	REPEATABLE	I
O	18	98	VANILLA	6F	4F	EF	CF	0F	0F	8F	8F	CAPSABLE	REPEATABLE	O
P	19	99	VANILLA	70	50	F0	D0	10	10	90	90	CAPSABLE	REPEATABLE	P
[{	1A	9A	VANILLA	5B	7B	DB	FB	1B	1B	9B	9B		REPEATABLE	[{
] }	1B	9B	VANILLA	5D	7D	DD	FD	1D	1D	9D	9D		REPEATABLE	] }
[undefined]	1C	9C												[undefined]
Num 1	1D	9D	VANILLA	31	31	B1	B1	31	31	B1	B1		REPEATABLE	Num 1
Num 2	1E	9E	VANILLA	32	32	B2	B2	32	32	B2	B2		REPEATABLE	Num 2
Num 3	1F	9F	VANILLA	33	33	B3	B3	33	33	B3	B3		REPEATABLE	Num 3

Table 1 (contd.)

Key Legend	"Raw" key codes key press	key release	key type	"Cooked" key codes alone	with SHIFT	with ALT	with SHIFT & ALT	with CTRL	with SHIFT & CTRL	with ALT & CTRL	with SHIFT, ALT & CTRL	whether capsable	whether repeatable	Key Legend
A	20	A0	VANILLA	61	41	E1	C1	01	01	81	81	CAPSABLE	REPEATABLE	A
S	21	A1	VANILLA	73	53	F3	D3	13	13	93	93	CAPSABLE	REPEATABLE	S
D	22	A2	VANILLA	64	44	E4	C4	04	04	84	84	CAPSABLE	REPEATABLE	D
F	23	A3	VANILLA	66	46	E6	C6	06	06	86	86	CAPSABLE	REPEATABLE	F
G	24	A4	VANILLA	67	47	E7	C7	07	07	87	87	CAPSABLE	REPEATABLE	G
H	25	A5	VANILLA	68	48	E8	C8	08	08	88	88	CAPSABLE	REPEATABLE	H
J	26	A6	VANILLA	6A	4A	EA	CA	0A	0A	8A	8A	CAPSABLE	REPEATABLE	J
K	27	A7	VANILLA	6B	4B	EB	CB	0B	0B	8B	8B	CAPSABLE	REPEATABLE	K
L	28	A8	VANILLA	6C	4C	EC	CC	0C	0C	8C	8C	CAPSABLE	REPEATABLE	L
; :	29	A9	VANILLA	3B	3A	BB	BA	3B	3A	BB	BA		REPEATBALE	; :
' "	2A	AA	VANILLA	27	22	A7	A2	27	22	A7	A2		REPEATABLE	' "
[reserved]	2B	AB												[reserved]
[undefined]	2C	AC												[undefined]
Num 4	2D	AD	VANILLA	34	34	B4	B4	34	34	B4	B4		REPEATABLE	Num 4
Num 5	2E	AE	VANILLA	35	35	B5	B5	35	35	B5	B5		REPEATABLE	Num 5
Num 6	2F	AF	VANILLA	36	36	B6	B6	36	36	B6	B6		REPEATABLE	Num 6
[reserved]	30	B0												[reserved]
Z	31	B1	VANILLA	7A	5A	FA	DA	1A	1A	9A	9A	CAPSABLE	REPEATABLE	Z
X	32	B2	VANILLA	78	58	F8	D8	18	18	98	98	CAPSABLE	REPEATABLE	X
C	33	B3	VANILLA	63	43	E3	C3	03	03	83	83	CAPSABLE	REPEATABLE	C
V	34	B4	VANILLA	76	56	F6	D6	16	16	96	96	CAPSABLE	REPEATABLE	V
B	35	B5	VANILLA	62	42	E2	C2	02	02	82	82	CAPSABLE	REPEATABLE	B
N	36	B6	VANILLA	6E	4E	EE	CE	0E	0E	8E	8E	CAPSABLE	REPEATABLE	N
M	37	B7	VANILLA	6D	4D	ED	CD	0D	0D	8D	8D	CAPSABLE	REPEATABLE	M
, <	38	B8	VANILLA	2C	3C	AC	BC	2C	3C	AC	BC		REPEATABLE	, <
. >	39	B9	VANILLA	2E	3E	AE	BE	2E	3E	AE	BE		REPEATABLE	. >
/ ?	3A	BA	VANILLA	2F	3F	AF	BF	2F	3F	AF	BF		REPEATABLE	/ ?
[undefined]	3B	BB											REPEATABLE	[undefined]
Num .	3C	BC	VANILLA	2E	2E	AE	AE	2E	2E	AE	AE		REPEATABLE	Num .
Num 7	3D	BD	VANILLA	37	37	B7	B7	37	37	B7	B7		REPEATABLE	Num 7
Num 8	3E	BE	VANILLA	38	38	B8	B8	38	38	B8	B8		REPEATABLE	Num 8
Num 9	3F	BF	VANILLA	39	39	B9	B9	39	39	B9	B9		REPEATABLE	Num 9

Table 1 (contd.)

Key Legend	"Raw" key codes: key press	"Raw" key codes: key release	"Cooked" key codes: key type	alone	with SHIFT	with ALT	with SHIFT & ALT	with CTRL	with SHIFT & CTRL	with ALT & CTRL	with SHIFT, ALT & CTRL	whether capsable	whether repeatable	Key Legend
Space	40	C0	ALT	20	20	A0	A0	20	20	A0	A0		REPEATABLE	Space
Backspace	41	C1	NOQUAL	08	08	08	08	08	08	08	08		REPEATABLE	Backspace
Tab	42	C2	SHIFT	09	9B5A	09	9B5A	09	9B5A	09	9B5A		REPEATABLE	Tab
Enter	43	C3	NOQUAL	0D	0D	0D	0D	0D	0D	0D	0D			Enter
Return	44	C4	CTRL	0D	0D	0D	0D	0A	0A	0A	0A			Return
Esc	45	C5	ALT	1B	1B	9B	9B	1B	1B	9B	9B			Esc
Del	46	C6	NOQUAL	7F	7F	7F	7F	7F	7F	7F	7F		REPEATABLE	Del
[undefined]	47	C7												[undefined]
[undefined]	48	C8												[undefined]
[undefined]	49	C9												[undefined]
Num -	4A	CA	ALT	2D	2D	FF	FF	2D	2D	FF	FF		REPEATABLE	Num -
[undefined]	4B	CB												[undefined]
Up	4C	CC	SHIFT	9B41	9B54	9B41	9B54	9B41	9B54	9B41	9B54		REPEATABLE	Up
Down	4D	CD	SHIFT	9B42	9B53	9B42	9B53	9B42	9B53	9B42	9B53		REPEATABLE	Down
Right	4E	CE	SHIFT	9B43	9B2040	9B43	9B2040	9B43	9B2040	9B43	9B2040		REPEATABLE	Right
Left	4F	CF	SHIFT	9B44	9B2041	9B44	9B2041	9B44	9B2041	9B44	9B2041		REPEATABLE	Left
F1	50	D0	SHIFT	9B307E	9B31307E	9B307E	9B31307E	9B307E	9B31307E	9B307E	9B31307E		REPEATABLE	F1
F2	51	D1	SHIFT	9B317E	9B31317E	9B317E	9B31317E	9B317E	9B31317E	9B317E	9B31317E		REPEATABLE	F2
F3	52	D2	SHIFT	9B327E	9B31327E	9B327E	9B31327E	9B327E	9B31327E	9B327E	9B31327E		REPEATABLE	F3
F4	53	D3	SHIFT	9B337E	9B31337E	9B337E	9B31337E	9B337E	9B31337E	9B337E	9B31337E		REPEATABLE	F4
F5	54	D4	SHIFT	9B347E	9B31347E	9B347E	9B31347E	9B347E	9B31347E	9B347E	9B31347E		REPEATABLE	F5
F6	55	D5	SHIFT	9B357E	9B31357E	9B357E	9B31357E	9B357E	9B31357E	9B357E	9B31357E		REPEATABLE	F6
F7	56	D6	SHIFT	9B367E	9B31367E	9B367E	9B31367E	9B367E	9B31367E	9B367E	9B31367E		REPEATABLE	F7
F8	57	D7	SHIFT	9B377E	9B31377E	9B377E	9B31377E	9B377E	9B31377E	9B377E	9B31377E		REPEATABLE	F8
F9	58	D8	SHIFT	9B387E	9B31387E	9B387E	9B31387E	9B387E	9B31387E	9B387E	9B31387E		REPEATABLE	F9
F10	59	D9	SHIFT	9B397E	9B31397E	9B397E	9B31397E	9B397E	9B31397E	9B397E	9B31397E		REPEATABLE	F10
[undefined]	5A	DA												[undefined]
[undefined]	5B	DB												[undefined]
[undefined]	5C	DC												[undefined]
[undefined]	5D	DD												[undefined]
[undefined]	5E	DE												[undefined]
Help	5F	DF	NOQUAL	9B3F7E	9B3F7E	9B3F7E	9B3F7E	9B3F7E	9B3F7E	9B3F7E	9B3F7E			Help

Table 1 (contd.)

Key Legend	"Raw" key codes		"Cooked" key codes											Key Legend
	key press	key release	key type	alone	with SHIFT	with ALT	with SHIFT & ALT	with CTRL	with SHIFT & CTRL	with ALT & CTRL	with SHIFT, ALT & CTRL	whether capsable	whether repeatable	
Left Shift	60	E0												Left Shift
Right Shift	61	E1												Right Shift
Caps Lock	62	E2												Caps Lock
Ctrl	63	E3												Ctrl
Left Alt	64	E4												Left Alt
Right Alt	65	E5												Right Alt
Left Amiga	66	E6												Left Amiga
Right Amiga	67	E7												Right Amiga

Intuition Direct Communication Message Ports (IDCMPs)

Application programs may obtain input event data by opening an Intuition window with IDCMP flags set (as in Figure 1(c)). Intuition's input event handler will then inform the application of events in which it is interested by sending it IntuiMessages describing the events. These contain data describing the events either in "raw" or processed form, in a way similar to that used by the "input event" structure. Raw mouse button events will be processed by Intuition's input event handler (to produce "cooked" mouse events), and the application will be informed that a gadget or menu has been selected or whatever.

It is also possible to get keyboard input via IDCMPs, using one of two flags:-

(1) Setting the RAWKEY IDCMP flag will cause Intuition to inform you of key presses and releases, using raw keycodes.

(2) Setting the VANILLAKEY IDCMP flag will cause Intuition to "cook" the keyboard input for you, giving you the data in ASCII form. To cook the data for you, Intuition makes use of one of the console device's library functions, RawKeyConvert(), which uses the current keymap to cook the raw data. Intuition only provides a single character output buffer for use by RawKeyConvert(), so keys such as the function keys, which generate more than one character, will not be passed on to you by Intuition. Thus you cannot use this method if you wish to know about cursor keys, function keys, etc - though you can set the RAWKEY flag then call RawKeyConvert() yourself (see Appendix 1).

The example program VANILLAKEY shows how to get "cooked" keyboard input by using the VANILLAKEY IDCMP option. If you run this program from a CLI and then press keys, you will get the "cooked" keycodes in hex displayed in the CLI window. Use CTRL-C to exit.

Using the console device directly

The most powerful and versatile way of handling keyboard input also requires the most work from the programmer to set it up. This involves opening an Intuition window and then "attaching" a console device to it, by using OpenDevice() with an appropriate IO Request block (see Figure 1(d)). This causes the console device's input event handler to be linked into the input handler chain after Intuition, and makes the given window the output window for the console device. Keyboard (and other) input events can then be obtained, normally in "cooked" form, by doing CMD_READs from the console device. Text output, scrolling, cursor movement, and so on are performed by CMD_WRITE commands to the console device, as are commands specifying that you want "raw" keycodes or other kinds of events to be reported to you. All special commands to the console use CSI sequences, with the form described above.

This approach has the following advantages:-

(1) the window can be opened on any screen, with any flags and whatever system or custom gadgets you may require;

(2) all keys can be obtained in "cooked" form, including cursor and function keys, etc;

(3) you can supply your own custom keymap and the console device will give you keyboard data cooked according to your recipe.

The example program CONSOLE illustrates this method. Like the VANILLAKEY program, it obtains "cooked" data and prints the values in hex in the CLI window used to invoke it, and exits when you press CTRL-C. However this program also illustrates the console device being used for output, and you get the text, cursor movement, etc that you specify occurring in the output window.

AmigaDOS "devices"

The simplest, but most limited, method of getting keyboard input is to make use of AmigaDOS "devices" (not to be confused with Exec devices such as the keyboard, gameport, input and console devices!!). These are accessed in the same way as AmigaDOS files, using DOS library calls Open(), Close(), Read(), Write(), etc, but are distinguished from files by having special names. Those used for obtaining keyboard input are "RAW:" and "CON:". These offer a shortcut method for opening a window and attaching a console device, but with a number of limitations:-

(1) the window will be opened on the Workbench screen - opening DOS windows on other screens cannot be done without special trickery (see the Fish disks);

(2) although the window's title and initial size and position can be specified, the programmer has no control over the maximum and minimum size of the window, and the sizing, push, pop and drag gadgets are always present, but no close gadget;

(3) no custom gadgets can be attached to the window;

(4) the default keymap (as set by SetMap) is used for key cookery - it is not possible to use a custom keymap.

There are in fact three ways of obtaining keyboard data with this method:-

(1) using Open("RAW:.....",MODE_NEWFILE) and then Read()ing from this "device" to obtain "cooked" data - this is illustrated by the example program DOSRAW (see Figure 1(e));

(2) using Open("RAW:.....",MODE_NEWFILE), Write()ing a control sequence to the "device" asking it to give us "raw" keycodes, then Read()ing from the "device" to obtain "raw" keycodes;

(3) using Open("CON:.....",MODE_NEWFILE) and then Read()ing from the "device" to obtain "cooked" data a line at a time, with backspace, etc handled transparently by the "CON:", and special keys such as cursor and function keys being suppressed.

The DOSRAW program is an example of using an AmigaDOS "RAW:" device for keyboard input and text output. It acts just like CONSOLE program, but involves much less programming effort. Note that the CONSOLE program has been written to behave as much as possible like this one: in particular, the maximum and minimum sizes of the window and the choice of system gadgets used are those which you always get with AmigaDOS "RAW:" or "CON:" devices.

Ten different ways to get user input

There are thus at least ten different ways to get user input:-

(1) directly access the keyboard & gameport devices;

(2) hook into the input event handler chain ahead of Intuition, so that you just get raw input events;

(3) hook into the input event handler chain behind Intuition, so that you only have to process events not dealt with by Intuition;

(4) open an Intuition window and use the RAWKEY IDCMP option - convert these into "cooked" data by using the console.device RawKeyConvert() function;

(5) open an Intuition window and use the VANILLAKEY IDCMP option;

(6) open an Intuition window and attach a console device, then get "cooked" data;

(7) open an Intuition window and attach a console device, then get "raw" data;

(8) open an AmigaDOS "RAW:" device and select "raw" input events;

(9) open an AmigaDOS "RAW:" device and get "cooked" data;

(10) open an AmigaDOS "CON:" device.

Those likely to prove most useful are:-

(4) for applications using IDCMPs for handling gadget selection and so on, but also requiring use of the keyboard;

(5) for applications using IDCMPs, requiring only limited use of the keyboard, avoiding the cursor and function keys, etc;

(6) for applications requiring heavy use of console IO, where AmigaDOS "RAW:" and "CON:" devices are not sufficient, eg because you are not working on the Workbench screen;

(9) for applications running on the Workbench screen, with less programming effort than using method (6);

(10) for inputting lines of text, with little control over formatting and no need for special keys.

Hopefully, by the time that you've digested the above, you should be in a position to make up your own mind about how you like your input events (including whether you prefer your mice raw or cooked!).

Appendix - More about RawKeyConvert()

Since the articles above and the examples which follow first appeared in Kickstart journal, there seems to have developed a consensus of opinion between Amiga programmers as to what input methods are generally most worth going for. While there will always be cases where special methods are required, the following is probably a good summary of current opinion:

(1) Unless you are doing something pretty trivial, or converting from a very file-oriented system such as Unix, there is not much to be said for bothering with the DOS CON: and RAW: devices.

(2) If you are writing a very text-oriented application where you want a standard ANSI terminal, open console.device and handle everything through this, including getting mouse events etc.

(3) In all other cases, the most versatile method is number (4) above - use an Intuition IDCMP, and perform any keycode conversion yourself using console.device RawKeyConvert().

Using RawKeyConvert() is reasonably straightforward. First of all you have to open a window with non-null IDCMP flags as in the VANILLAKEY example which follows; however you should set RAWKEY rather than VANILLAKEY in the IDCMPFlags field of your NewWindow structure. Next, you need to open console.device with a special unit number -1, which indicates that you only want to use it as a library; you need a IOStdReq structure lying around somewhere in order to do this, but you **don't** need all the special setting up of ports etc showed in the CONSOLE example below. Other stuff you will need are an InputEvent structure initialised as shown below, and a buffer for storing converted key-codes:

```
/* global data used by key conversion */

struct Device *ConsoleDevice;

struct IOStdReq ConsoleReq;
struct InputEvent RawKeyEvent = {NULL,IECLASS_RAWKEY,0,0,0};

#define BUFSIZ 10
UBYTE KeyBuffer[BUFSIZ];
```

The console open is then simply as follows:

```
if (OpenDevice("console.device",-1,&ConsoleReq,0)) {
        /* gone wrong - cope with failure! */
}

ConsoleDevice = ConsoleReq.io_Device;   /* used as library
                                           base address */
```

When you get an IDCMP message with class RAWKEY, you can now call RawKeyConvert() as follows:

```
      struct IntuiMessage *IDCMPMsg;
      LONG KeyCodes;

      /* (get info from IDCMP in usual way here) */

      if (IDCMPMsg->Class == RAWKEY) {   /* now process key events */

            RawKeyEvent.ie_Code = IDCMPMsg->Code;
            RawKeyEvent.ie_Qualifier = IDCMPMsg->Qualifier;
            RawKeyEvent.ie_position.ie_addr = NULL;
            KeyCodes = RawKeyConvert(&RawKeyEvent,KeyBuffer,BUFSIZ,NULL);

            if (KeyCodes > 0) {
                                    /* process key codes from buffer */
            }
      }
```

What is going on here is that we are picking up information from our IDCMP message and using it to "reconstruct" a RAWKEY input event as might be passed to the console device as part of the input device's chain of event handlers. This is a bit round the houses, but there you go. We then call RawKeyConvert(); picking up the library base address from location ConsoleDevice will be handled by the "stub" routine from amiga.lib in the usual manner. The value returned - KeyCodes in our example - tells us how many converted key codes are now available in KeyBuffer. This might be zero if the raw key event being processed was just shift going down (say), one if the key hit was an ordinary alphanumeric, more than one - ie a CSI sequence - if the key was a function key etc - or minus one if your buffer overflowed!

Note that we are calling RawKeyConvert with a NULL final parameter, which tells it to use the console device's default keymap, as set up by the utility program SetMap. Alternatively, you could pass it the address of your own KeyMap structure.

Finally, note that this procedure needs some minor adjustment if we want to handle the "dead keys" now available in some European keymaps; these are keys which have no visible effect when hit, but cause the next character output to be modified or accented in some way. To get this right, we make use of the IAddress field from our IntuiMessage, by setting up

```
RawKeyEvent.ie_position.ie_addr = *((APTR *)IDCMPMsg->IAddress);
```

before calling RawKeyConvert().

Finally, we must of course close everything down when we exit; in the case of console.device, this is handled in the normal way, by

```
      CloseDevice(&ConsoleReq);
```

```
/* --------------------- VANILLAKEY --------------------
   Example of obtaining 'cooked' keyboard input by opening
   an Intuition window and using IDCMP VANILLAKEY messages
   ------------------------------------------------------ */

#include <exec/types.h>
#include <intuition/intuition.h>

extern APTR OpenLibrary();                      /* Exec library */
extern VOID CloseLibrary();
extern struct Message *WaitPort(),*GetMsg();
extern VOID ReplyMsg();

extern struct Window *OpenWindow();             /* Intuition library */
extern VOID CloseWindow();

extern VOID printf();                           /* Amiga.lib */

/*** Variables ***/

APTR IntuitionBase = NULL;                      /* Intuition library base address */

struct Window *ConsoleWindow = NULL;            /* ptr to console window */

UBYTE HexString[3];                             /* hex string to output */
UBYTE HexChar[] = "0123456789ABCDEF";

/*** Definition of console window ***/

static struct NewWindow ConNewWindow = {

   400,30,                                      /* LeftEdge, TopEdge */
   200,80,                                      /* Width, Height */
   -1,-1,                                       /* DetailPen, BlockPen */
   VANILLAKEY,                                  /* IDCMPFlags */
   WINDOWDEPTH | WINDOWDRAG | WINDOWSIZING
   | SMART_REFRESH | ACTIVATE,                  /* Flags */
   NULL,                                        /* FirstGadget */
   NULL,                                        /* CheckMark */
   "Console window",                            /* Title */
   NULL,                                        /* Screen */
   NULL,                                        /* BitMap */
   120,50,                                      /* MinWidth, MinHeight */
   640,200,                                     /* MaxWidth, MaxHeight */
   WBENCHSCREEN                                 /* Type */
};

/* Close window, etc & then exit */

VOID CleanUpAndExit()
{
   if (ConsoleWindow != NULL)
      CloseWindow(ConsoleWindow);

   if (IntuitionBase != NULL)
      CloseLibrary(IntuitionBase);

   Exit(TRUE);
}
```

```
/*** Open Intuition window with IDCMP VANILLAKEY input ***/

VOID Init()
{
   if ((IntuitionBase = OpenLibrary("intuition.library",29)) == NULL)
      CleanUpAndExit();

   if ((ConsoleWindow = OpenWindow(&ConNewWindow)) == NULL)
      CleanUpAndExit();

   return;
}

/*** Main program function ***/

main()
{
   struct IntuiMessage *message; /* pointer to message received */
   ULONG class;                  /* class of message being processed */
   USHORT code;                  /* code of message being processed */
   UBYTE c = '\0';               /* current character read from console */

   Init();               /* open Intuition window with VANILLAKEY IDCMP */

   do {
      WaitPort(ConsoleWindow->UserPort);

      while (message = (struct IntuiMessage *) GetMsg(ConsoleWindow->UserPort)) {

         class = message->Class;
         code = message->Code;

         ReplyMsg(message);

         if (class == VANILLAKEY) {
            c = code;
            HexString[0] = HexChar[c >> 4];
            HexString[1] = HexChar[c & 0x0F];
            HexString[2] = '\0';
            printf("%s",HexString);
         }
      }
   } while (c != '\003');

   printf("\n");

   CleanUpAndExit();
}

/*** The End ***/
```

```
/* ---------------------- CONSOLE ----------------------
   Example of obtaining 'cooked' keyboard input by opening
   an Intuition window and attaching a console device
   ------------------------------------------------------ */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <devices/console.h>

extern APTR OpenLibrary();                      /* Exec library */
extern LONG OpenDevice();
extern VOID DoIO();
extern VOID CloseDevice(),CloseLibrary();

extern struct MsgPort *CreatePort();            /* Exec support library */
extern struct IOStdReq *CreateStdIO();
extern VOID DeletePort(),DeleteStdIO();

extern struct Window *OpenWindow();             /* Intuition library */
extern VOID CloseWindow();

extern VOID printf();                           /* Amiga.lib */

/*** Variables ***/

APTR IntuitionBase = NULL;                      /* Intuition library base address */

struct Window *ConsoleWindow = NULL;            /* ptr to console window */
struct MsgPort *ConWrtPort = NULL;              /* ptr to console write message port
struct MsgPort *ConReadPort = NULL;             /* ptr to console read message port *
struct IOStdReq *ConWrtReq = NULL;              /* ptr to console write request block
struct IOStdReq *ConReadReq = NULL;             /* ptr to console read request block

LONG ConsoleOpen = FALSE;                       /* flags whether console open */

#define CONREADBUFLEN   80                      /* length of console read buffer */
UBYTE ConReadBuffer[CONREADBUFLEN];             /* console read buffer */

UBYTE HexString[2*CONREADBUFLEN+1];             /* hex string to output */
UBYTE HexChar[] = "0123456789ABCDEF";

/*** Definition of console window ***/

static struct NewWindow ConNewWindow = {

   400,30,                                      /* LeftEdge, TopEdge */
   200,80,                                      /* Width, Height */
   -1,-1,                                       /* DetailPen, BlockPen */
   0,                                           /* IDCMPFlags */
   WINDOWDEPTH | WINDOWDRAG | WINDOWSIZING
   | SMART_REFRESH | ACTIVATE,                  /* Flags */
   NULL,                                        /* FirstGadget */
   NULL,                                        /* CheckMark */
   "Console window",                            /* Title */
   NULL,                                        /* Screen */
   NULL,                                        /* BitMap */
   120,50,                                      /* MinWidth, MinHeight */
   640,200,                                     /* MaxWidth, MaxHeight */
   WBENCHSCREEN                                 /* Type */
```

```
};

/* Close console device, window, etc & then exit */

VOID CleanUpAndExit()
{
   if (ConsoleOpen)
      CloseDevice(ConWrtReq);

   if (ConReadReq != NULL)
      DeleteStdIO(ConReadReq);

   if (ConWrtReq != NULL)
      DeleteStdIO(ConWrtReq);

   if (ConReadPort != NULL)
      DeletePort(ConReadPort);

   if (ConWrtPort != NULL)
      DeletePort(ConWrtPort);

   if (ConsoleWindow != NULL)
      CloseWindow(ConsoleWindow);

   if (IntuitionBase != NULL)
      CloseLibrary(IntuitionBase);

   Exit(TRUE);
}

/*** Open Intuition window & attach console device ***/

VOID Init()
{
   if ((IntuitionBase = OpenLibrary("intuition.library",29)) == NULL)
      CleanUpAndExit();

   if ((ConsoleWindow = OpenWindow(&ConNewWindow)) == NULL)
      CleanUpAndExit();

   if ((ConWrtPort = CreatePort(0,0)) == NULL)
      CleanUpAndExit();

   if ((ConReadPort = CreatePort(0,0)) == NULL)
      CleanUpAndExit();

   if ((ConWrtReq = CreateStdIO(ConWrtPort)) == NULL)
      CleanUpAndExit();

   if ((ConReadReq = CreateStdIO(ConReadPort)) == NULL)
      CleanUpAndExit();

   ConWrtReq->io_Data = (APTR) ConsoleWindow;
   ConWrtReq->io_Length = sizeof(struct Window);

   if (OpenDevice("console.device",0,ConWrtReq,0) != 0)
      CleanUpAndExit();
   else
      ConsoleOpen = TRUE;
```

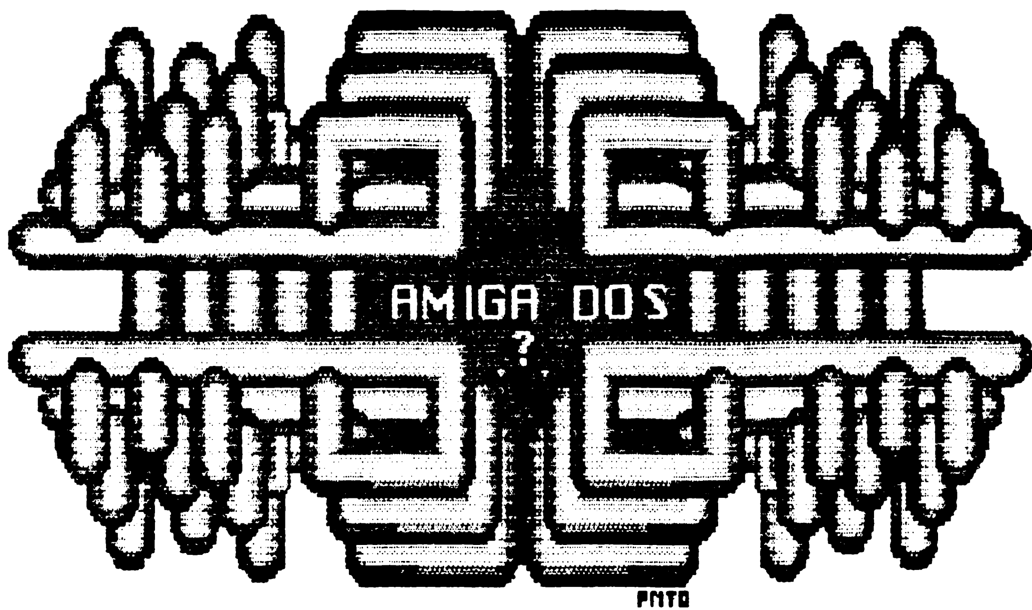
```
   ConReadReq->io_Device = ConWrtReq->io_Device;
   ConReadReq->io_Unit = ConWrtReq->io_Unit;

   return;
}

/*     Read characters from console     *
 *  (returns number of characters read) */

ULONG ReadConsole(buffer,buflen)
STRPTR buffer;
ULONG buflen;
{
   ConReadReq->io_Data = (APTR) buffer;
   ConReadReq->io_Length = buflen;
   ConReadReq->io_Command = CMD_READ;

   DoIO(ConReadReq);                 /* wait for character(s) from console */

   return(ConReadReq->io_Actual);    /* return number of chars read */
}

/*** Write string of specified length to console ***/

VOID WrtConsole(string,length)
STRPTR string;
ULONG length;
{
   ConWrtReq->io_Data = (APTR) string;
   ConWrtReq->io_Length = length;
   ConWrtReq->io_Command = CMD_WRITE;

   DoIO(ConWrtReq);
}

/*** Main program function ***/

main()
{
   ULONG n;              /* number of characters read */
   ULONG i;              /* index to current character in read buffer */
   ULONG j;              /* index to character position in output string */
   UBYTE c;              /* current character read from console */

   Init();               /* open Intuition window & attach console device */

   do {
      n = ReadConsole(ConReadBuffer,CONREADBUFLEN);

      WrtConsole(ConReadBuffer,n);

      for (i = 0, j = 0; i < n; i++) {
         c = ConReadBuffer[i];
         HexString[j++] = HexChar[c >> 4];
         HexString[j++] = HexChar[c & 0x0F];
      }
      HexString[j++] = '\0';
      printf("%s",HexString);

   } while (c != '\003');
```

```
    printf("\n");

    CleanUpAndExit();
}

/*** The End ***/
```

```
/* ----------------------- DOSRAW -----------------------
   Example of obtaining 'cooked' keyboard input by opening
   an AmigaDOS "RAW:" file
   ------------------------------------------------------- */

#include <exec/types.h>
#include <libraries/dos.h>

extern APTR Open();                            /* DOS library */
extern VOID Close();
extern ULONG Read(),Write();

extern VOID printf();                          /* Amiga.lib */

/*** Variables ***/

APTR ConsoleFile = NULL;                       /* console file handle */

#define CONREADBUFLEN   80                     /* length of console read buffer */
UBYTE ConReadBuffer[CONREADBUFLEN];            /* console read buffer */

UBYTE HexString[2*CONREADBUFLEN+1];            /* hex string to output */
UBYTE HexChar[] = "0123456789ABCDEF";

/* Close file & then exit */

VOID CleanUpAndExit()
{
   if (ConsoleFile != NULL)
      Close(ConsoleFile);

   Exit(TRUE);
}

/*** Open AmigaDOS "RAW:" file ***/

VOID Init()
{
   if ((ConsoleFile = Open("RAW:400/30/200/80/Console window",MODE_NEWFILE))
         == NULL)
      CleanUpAndExit();

   return;
}

/*     Read characters from console      *
 *  (returns number of characters read) */

ULONG ReadConsole(buffer,buflen)
STRPTR buffer;
ULONG buflen;
{
   return(Read(ConsoleFile,buffer,buflen));
}

/*** Write string of specified length to console ***/

VOID WrtConsole(string,length)
STRPTR string;
ULONG length;
```

```
{
   Write(ConsoleFile,string,length);
}

/*** Main program function ***/

main()
{
   ULONG n;              /* number of characters read */
   ULONG i;              /* index to current character in read buffer */
   ULONG j;              /* index to character position in output string */
   UBYTE c;              /* current character read from console */

   Init();               /* open AmigaDOS "RAW:" file */

   do {
      n = ReadConsole(ConReadBuffer,CONREADBUFLEN);

      WrtConsole(ConReadBuffer,n);

      for (i = 0, j = 0; i < n; i++) {
         c = ConReadBuffer[i];
         HexString[j++] = HexChar[c >> 4];
         HexString[j++] = HexChar[c & 0x0F];
      }
      HexString[j++] = '\0';
      printf("%s",HexString);

   } while (c != '\003');

   printf("\n");

   CleanUpAndExit();
}

/*** The End ***/
```

Aspects of AmigaDOS



"Keys" illustration by Paula Dawson.

Section 4 - ABC AmigaDOS

The Technical Reference section of the AmigaDOS Manual is full of useful information - it is not however an easy read by any standards! For one thing, the information is densely packed; for another it uses a lot of technical terms which may or may not be familiar. This section aims to help by explaining some of the important terms in a way which is supposed to be illuminating rather than strictly rigorous. Like AmigaDOS itself, this section is designed either for random or sequential access - an alphabetic "key table" is provided, followed by a list of topics, which have been arranged to make sense when read sequentially.

When reading this section, be particularly wary of the fact that some important terms such as "library" and "device" are used to mean (at least) one thing by AmigaDOS, and something else by Exec! In particular, "device" in this section will usually be used to refer to an AmigaDOS "device" such as "DF0:" - this is NOT the same as an Exec "device" such as "trackdisk.device", as discussed in the previous section. We shall occasionally mention trackdisk.device - remember that devices in this sense are a lower level aspect of the system than most of what we are discussing, and are a part of Exec on which AmigaDOS is built.

Missing from this section are a number of important terms concerned with binary file format and linking - these include "hunk", "block", "program unit", etc. For an explanation of this material, see part two of the section about "libraries" on Amiga.

Bear in mind AmigaDOS history when reading this section. In summary, the Amiga was originally supposed to have a "file system and process manager" interacted with the rest of the operating environment. When this project got into trouble - Amiga subcontracted it to another company - Metacomco were asked to put the Tripos operating system on Amiga; the original port of Tripos was done by Tim King and his team in about a month, which wasn't bad going!

Tripos is written in a language called BCPL, similar to C but with some important differences. Its implementation on Amiga uses Exec as its "kernel", and Exec devices such as trackdisk.device as its "device drivers". Tripos sits very comfortably on Exec, since the two systems follow very much the same basic design principles. It sits less comfortably on trackdisk; Tripos was designed for big hard-disk multi-user systems, and tends to sacrifice speed to versatility and recoverability, whereas trackdisk is written for speed. In some circumstances this can result in the worst of both worlds, with Tripos slowness combined with trackdisk lack of recoverability! However, this is by no means always the case - AmigaDOS is at its slowest performing operations like DIR or LIST when it doesn't yet know filenames, faster finding **named** files, and VERY fast once it has files open - see below for further discussion.

KEY TABLE

BCPL	2.1	Header key	1.11
Bitmap	1.8	IO Stream	3.2
Block	1.2	Kernel	3.1
Block list	1.5	Key	1.11
BPTR	2.3	Lock	1.15
BSTR	2.4	Martian	2.1
Cache buffers	1.13	Redundancy	1.7
Checksum	1.10	Resident library	2.6
CLI	3.5	Resident segment	2.5
Co-routine	3.4	Root	1.4
Command	3.6	Packet	3.3
Cylinder	1.3	Parent	1.4
Device	3.8	Path	1.4
Device	3.8	Priority	3.2
Device	3.8	Process	3.2
Directory	1.4	Protection bits	1.16
DOS library	3.1	Redirection	3.7
Extension	1.5	Secondary type	1.5
File handler	1.1	Segment list	2.5
Filing system	1.1	Sequence number	1.12
Grand Bodge	3.3	Shell	3.5
Global vector	2.2	Stack	3.9
Handle	1.14	Token passing	1.15
Handler	3.9	Tree	1.4
Hash chain	1.9	Type	1.5
Hash function	1.9	User directory	1.4
Hash number	1.9	Validate	1.8
Hash table	1.9	Volume	1.6
Header	1.11		

1. AMIGADOS FILING SYSTEM

1.1 Filing system/File handler

"Filing system" is a general term for that part of AmigaDOS which is concerned with files and directories; "file handler" is a specific name for an AmigaDOS "handler process" associated with an AmigaDOS "device" (such as DF0:), whose job is to look after filing. File handlers can be thought of as organised as a number of "layers" or "levels" - the low levels talk directly to trackdisk.device and work in terms of tracks and sectors; the higher levels abstract away from this and deal in terms of sequentially numbered "blocks". An intermediate level deals with translating between logical block numbers and actual physical tracks, sectors and surfaces.

1.2 Block

A basic unit of filing information, as viewed by the higher levels of AmigaDOS; 512 bytes of information distinguished by a unique "block number". From the point of view of the higher levels of AmigaDOS, a floppy disk appears simply as a series of blocks with "logical block numbers" from zero to 1759 - other disks such as hard disk or non-standard floppies appear very similar, with different maximum block numbers.

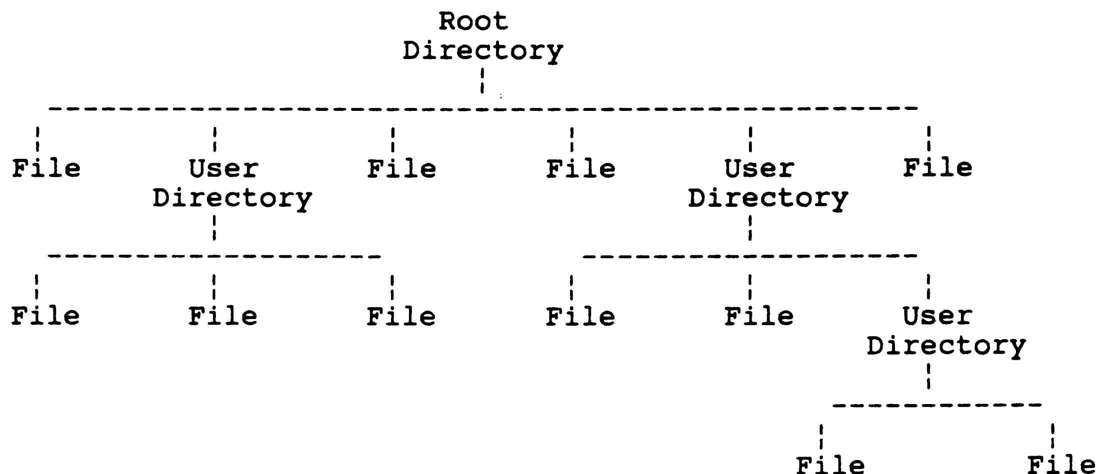
1.3 Cylinder

Originally a term from those huge mainframe disk units, consisting of a pack of hard disks with a disk head for each surface; the heads move together, and the area of every disk currently under a head is known collectively as a "cylinder". On a double-sided Amiga floppy with two surfaces and two heads, also moving together, sectors 0 to 10 on one disk surface plus the corresponding sectors on the other side are known collectively as a "cylinder", with sectors 0 to 21.

AmigaDOS thinks in terms of cylinders when translating between logical blocks and physical tracks and sectors - thus blocks 0 to 10 correspond to track 0 sectors 0 to 10 on one side of the disk, and blocks 11 to 21 are found on the same cylinder sectors 11 to 21, ie track 0 on the other disk surface; block 22 follows on track 1 sector 0 on the first side, etc. At first sight this seems peculiar - however, given that the heads move together, a moment's thought should convince you that this arrangement is likely to result in much less head movement than the one more often found on micros, in which tracks 0 to 79 are on one side of the disk and tracks 80 to 159 on the other. AmigaDOS is therefore sensible in this respect; other systems are less efficient.

1.4 Tree

One of the fundamental data structures of computer science (along with linked lists, stacks, queues and all that stuff) dearly beloved by Donald Knuth and good computer scientists everywhere. Computer trees tend to be upside down and to grow downwards (there must be some significance in this somewhere). The AmigaDOS "hierarchy" of directories is a "pure tree" as illustrated:



The root directory is created when the disk is formatted, and must always be present for the disk to be useable; user directories are created (surprise) by the user, usually by the DOS command MAKEDIR. The thing immediately above something in the tree is known as its "parent"; a route through the tree starting from the root and ending with a specific file or directory is known as a "path". Information about the root, including disk name, time of creation etc, is stored in a special block ST.ROOT - this is found at block number 880, ie cylinder 40 sector 0, ie track 40 sector 0 on the first disk surface. This is (sensibly) in the middle of the disk - hopefully out of harm's way - rather like the "directory" in old CBM DOS. Note however that the resemblence to old CBM DOS ends there - "directory blocks" other than the root can be found anywhere on the disk (though AmigaDOS now tries to keep them fairly close to each other to cut down directory search times). This means that it is impossible to suffer from a "full directory" unless the whole disk is full, which is rather clever.

1.5 Type/secondary type

Blocks are used to contain different types of filing information, and are distinguished by a "type" and a "secondary type" as follows:

Type	Secondary Type	Purpose
T.SHORT	ST.ROOT	"Root" of directory; information about entire "volume" (disk) - name, time of creation etc.
T.SHORT	ST.USERDIR	Information about a directory - name, comment, time of creation etc.
T.SHORT	ST.FILE	Information about a file - name, comment, etc, plus pointers to the disk blocks containing the actual data.
T.LIST	ST.FILE	"Extension" information, containing more pointers to data blocks for long files.
T.DATA	(none)	Actual data in file.

ST.ROOT and ST.USERDIR blocks contain "hash tables" containing up to 72 pointers to "hash chains" of ST.USERDIR or ST.FILE blocks. ST.FILE blocks contain "block lists" of up to 72 pointers to T.DATA blocks; if this isn't enough - ie if the file is over about 34K - then the ST.FILE block sets up an "extension" pointer to a T.LIST block containing more block list; this may also have an extension pointer if necessary, etc. See Fig 1 for a diagram showing the relationship between different block types, when accessing a file MYDISK:MYDIR/FRED.

1.6 Volume

For floppies, "volume" is just a fancy name for a disk, when viewed as a logical unit by AmigaDOS. Things may or may not be this simple for other media - eg a RAM disk is also viewed as a "volume", while with a hard disk, it may be convenient to split the physical disk into various separate logical volumes, ie to treat it like several floppies.

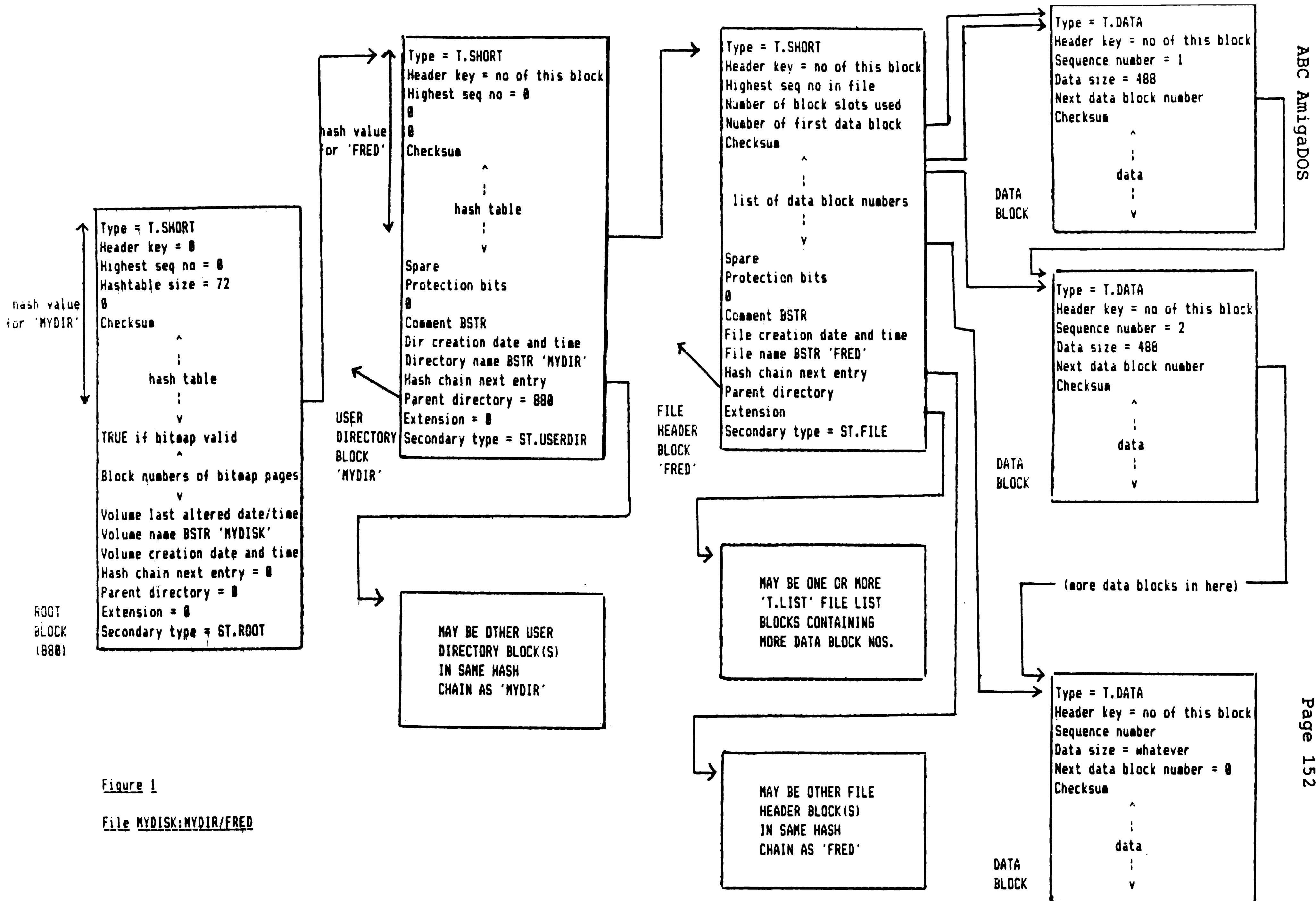


Figure 1

File MYDISK:MYDIR/FRED

1.7 Redundancy

Storing the same information in various different ways - ie redundantly. For example, information about how data blocks fit together to form a file is stored both as a series of pointers in ST.FILE file information blocks (a bit like CP/M), and as forwards pointers from one block to the next within the data blocks themselves (a bit like old CBM DOS sequential files). This allows you to "have your cake and eat it" - eg the pointers in ST.FILE blocks are more convenient for random access, while the internal pointers within the data blocks are handy for sequential access. Redundancy also helps recovery from errors - eg if a ST.FILE block gets splatted somehow, you can still (theoretically) put the file back together by looking at the internal data block pointers.

1.8 Bitmap

Information about which blocks in a volume are currently in use - bit set if block in use, clear if block available, just like old CBM DOS Block Availability Map (BAM). May or may not be "valid" on a given volume - AmigaDOS can check this by wizzing round the directory tree finding out which blocks are currently being used, rather like old CBM DOS "validate". DOS checks this whenever a disk is inserted - if it fails for some reason, DOS takes exception to the disk, and you're in trouble. Sort it out using DISKDOCTOR.

1.9 Hashing

Hashing is a technique used by AmigaDOS to make finding a given (named) file faster than having to search the whole directory.

Suppose AmigaDOS wants to find some given file "FRED" within the current directory. First of all, it applies a "hash function" to the file name FRED to generate a small "hash number" from it - a simple mechanism it might (but doesn't) use would be to add the ASCII values for 'F', 'R', 'E' and 'D' together, then divide the result down until it gets a small number in the right range (0 to 71). It then uses this small number as an offset in a "hash table" of pointers to ST.FILE file header blocks. It then searches through a (hopefully short) "hash chain" of header blocks for files with the same hash number, until it finds filename "FRED", or reaches the end of the hash chain and reports "File not found".

In fact, just adding the ASCII values for the characters together wouldn't be very clever, as this would tend to lead to "bunching" of files amongst the hash chains, and hence to longer than necessary search times. The mechanism actually used can be represented in pseudo-BASIC as follows:

```
hash = length of filename
for character = first to last
    hash = 13*(hash) + upper case ASCII value for character
next
```

(13 is chosen just as a nice prime number!). The result of this calculation is then rounded down mod 72 (the number of entries in the hash table) to produce an offset into the hash table in the file header block; six is then added onto this result to enable the offset to be applied as a longword offset from the start of the ST.FILE block rather than the start of the hash table itself, for reasons which escape the current author.

Checksum

How AmigaDOS tells if a block has gone bad on it - again, very like old CBM DOS. Each block contains a "checksum" arranged so that if all the longwords in the block are added together, and the result rounded down to the nearest longword, then the result should be zero - if not, the block has got corrupted somehow. It should be noted that using a simple sum like this isn't the most reliable form of error detection possible - it is however nice and simple, and has a much lower processor overhead than messing about with more sophisticated techniques like "Cyclic Redundancy" checkwords.

Key

Nothing much to do with a "lock"!

A "key" in AmigaDOS is simply a logical block number - when used to reference one block from another this is frequently referred to as a "pointer" in the documentation. For example, a ST.FILE contains a "block list" list of "data block keys" - this is just a list of the block numbers for the file's blocks of data. Most blocks have a "header key" immediately following the block type - this is the block's own logical block number.

Sequence number

A number stored in a data block to number the blocks in a file sequentially. Note that the data blocks are also ordered implicitly by the order they appear in the ST.FILE block list, and by the fact that each block contains a pointer to the next one - see "redundancy" above.

1.13 Cache buffers

Buffers used by AmigaDOS to remember ("cache") the last few segments read off disk, or to store up output material prior to writing to disk. Used to speed things up - eg if you dump a small file (eg TYPE .info OPT H) then immediately dump it again, you will get the second dump immediately with no disk access, because the file contents are currently in a cache buffer.

AmigaDOS cache buffers should not be confused with trackdisk device buffers, though the latter also serve to speed up disk access. When AmigaDOS asks trackdisk.device to read a particular sector, trackdisk.device actually reads the whole track into an internal buffer; requests for further sectors from the same track can then be satisfied without further disk access. When trackdisk returns a sector to AmigaDOS, DOS will itself remember it in its own cache buffer, thus providing another level of buffering. The trackdisk.device buffers have to be in chip memory since trackdisk uses the blitter; AmigaDOS cache buffers do not need to be in chip memory, and will normally be allocated in fast memory if available.

AmigaDOS normally allocates 5 cache buffers; if you are using 1.2 and have lots of memory, then you can increase this for even faster disk access using the new command ADDBUFFERS.

1.14 Handle

A number used by AmigaDOS to identify a particular currently open file, a bit similar to a "logical file number" in old CBM DOS. In old CBM DOS (anyone remember it?) you might open "logical file" number 1 (say) to the floppy disk as follows:

```
OPEN 1,8,8,"wombat,s,r"
```

The file can then be accessed via its "logical file number", as in

```
GET#1, A$
```

In more recent filing systems (eg the BBC Micro) the number used to access the file is allocated by the system, rather than being chosen by the programmer, and is known as a "handle", eg

```
WOM_HANDLE = OPENIN("wombat")
A = BGET#(WOM_HANDLE)
```

In this case, the "handle" used to access the file will be a small number returned by the OPENIN function.

AmigaDOS works rather like this, except that the handle returned is in fact a BPTR to a structure containing various information about the file; usually you don't have to worry about this, you just treat the handle as a number used to identify the file:

```
ULONG     handle,charsread,Open(),Read();
UBYTE     character;

handle = Open("wombat",MODE_OLDFILE);
charsread = Read(handle,&character,1);
```

(This is lousy C, just as the above is lousy BASIC - in real life, we would of course need to check for all these calls failing!)

<u>Lock</u>

Nothing much to do with a "key"!

A lock is a data structure maintained by AmigaDOS - calls which return a "lock" actually return a pointer to this structure. Locks serve two purposes:

a. The primary purpose of locks is to provide a mechanism for file (not record) locking, when there is a possibility of contention between processes. Locks come in two flavours, shared read-locks and exclusive write-locks. As many processes as want one can have a read-lock on a file at a time, but while something is read-locked, it is not possible to get a write-lock on it. Only one process at a time can have a write-lock; while this is set, nothing else is allowed a read or write lock. It can be seen that this provides a simple way of avoiding a situation where one process is trying to read a file while another is in the process of updating it. A process which currently has a lock can make it available to another process, by sending that process a message containing a pointer to the lock; the other process can then indicate when it has finished by replying this message. This is referred to as "token passing". Note however that if two processes require record (rather than file) locking, they will have to sort it out between themselves, using some private message-passing protocol.

b. A secondary function of locks is to provide something a bit like file-handles, only different - a lock in this sense is a number used to describe something like a directory to AmigaDOS, much like handles describe open files. For example, it is possible to obtain a lock on the current directory by calling Lock() with a null directory name (this is undocumented!); it is then possible to examine this directory by calling Examine(lock,FileInfoBlock). Locks (rather than handles) are also used to describe files from the point of view of functions like Examine() which don't

actually need to get the file open - note that obtaining a lock on a file is much faster than opening it, so calling Lock() (then UnLock()) provides a quick method of finding out if a given file exists.

Two further points worth noting about AmigaDOS locks are as follows:

1. Locks on directories in sense (b) above don't function in sense (a) - ie just because you have a lock on a directory, you won't stop some other process updating it!

2. Opening a file <u>implies</u> a lock in sense (a) to AmigaDOS - opening a file MODE_OLDFILE implies a read-lock, so no other process (or you) can write-lock it; opening a file MODE_NEWFILE implies a write-lock, so no other process can get a read- or write-lock until you close it. A weakness of 1.0 and 1.1 was that it wasn't possible to get a write-lock on an existing file; 1.2 supports a new MODE_READWRITE which opens an existing file with a write-lock, while MODE_OLDFILE is now known synonymously as MODE_READONLY.

1.16 <u>Protection bits</u>

Four bit flags which indicate if the current file can be read (bit 3), written (bit 2), executed (bit 1) or deleted (bit 0). Due to an argument between AmigaDOS and the requirements of the rest of the system software, DOS itself only pays attention to bit 0 (deletion) - the other bits are there if you want them however, and it's up to you if you choose to pay any attention to them.

AMIGADOS AND BCPL

BCPL

BCPL is a systems programming language invented in 1967 (Sergeant Pepper - remember?) by Dr Martin Richards of Cambridge University Computing Laboratory. Most of the Tripos operating system was written in BCPL, so BCPL and Tripos bear much the same relationship to each other as C and Unix; since Tripos started around 1976, BCPL and Tripos have advanced together, so most BCPL development environments contain many Tripos-like features, including those irritating "templates". The design philosophy of both BCPL and Tripos is to keep things small, portable, and simple; C and Unix on the other hand are much bigger (C is descended from BCPL), not especially portable (Unix was not designed as a portable system!), and certainly not simple - Unix is closely associated with the concept of "Martians", defined as strange creatures with sixteen fingers who go around saying "GREP!" to one another. Which you prefer is a matter of taste - Tripos is "TRIvial Portable Operating System" to some of its detractors, which at least is more polite than TripeDOS!

BCPL was designed for interactive system software development, and contains a number of features designed to cut down time spent hanging around waiting for compilers and linkers (sound familiar?). BCPL normally compiles into an intermediate code, which is then interpreted at run-time, very much like many Pascal (P-code) systems; this leads to rapid compilation. Long link-times are avoided using a mechanism known as the "global vector". Note that we can find no trace of intermediate code in AmigaDOS - while this was probably used in the course of development, the final version has been compiled down to 68000.

Global Vector

To cut down link-times, BCPL allows modules to link at run-time; different modules communicate using a common data area, which is a stonking great table containing global variables and addresses of global routines that all modules know about. Since "stonking great table" doesn't sound very professional, this is known instead as a "global vector" - "vector" is generally used in a rather confusing way in BCPL, to simply mean "some chunk of memory". The different bits of AmigaDOS are linked together in this way, which is why you keep coming across pointers to a "global vector" in the documentation. Note that the global vector table is for the internal convenience of AmigaDOS in tying its different bits together - it is likely to vary from version to version, and application programs are not supposed to use it. Most references to "global vector" refer to one master table used by the whole of AmigaDOS; however some processes, such as the file-handler, are tied together using their own private global vector.

2.3 Words and BPTRs

C is a "weakly typed" language in which types can be converted by an operation called "casting" - BCPL goes further than this, by not being typed at all, or supporting only one data type, which comes to the same thing. The data-type supported by BCPL is the machine-word; in 68000 BCPL implementations this is taken to be a 32-bit quantity, or longword (LONG or ULONG). AmigaDOS makes use of normal machine addresses (APTRs) internally - for example entries in the global vector corresponding to global routines are usually APTRs, so that the routines can be accessed by mechanisms like

```
MOVEA.L   APTR,An
JSR       (An)
```

However, since BCPL does everything in terms of longwords, its own pointers are expressed in a different way, which is a machine address expressed in longwords, otherwise known as a BPTR, where

```
BPTR value = APTR value / 4
```

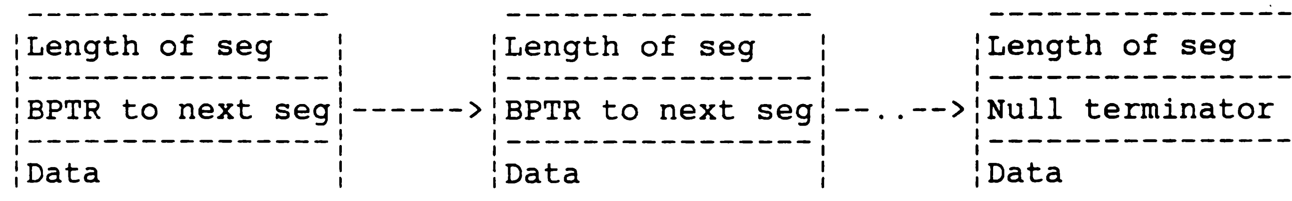
Note that anything accessed via a BPTR had better be longword aligned or there'll be trouble - this is something to watch out for when using AmigaDOS. Unfortunately, most of the rest of the Amiga software is written in C which tends to work with ordinary byte-oriented machine-addresses - the necessity of having to keep shifting addresses left or right by two when moving between AmigaDOS's BPTRs and the rest of the system's APTRs is something you get used to, but a nuisance.

2.4 Strings and BSTRs

BCPL also handles strings differently from C - C strings are pointers to a string of characters terminated by a null, while BCPL strings (BSTRs) are BPTRs to a structure containing the length of the string in the first byte, with the actual string characters following. Be warned that references to BSTRs normally mean a BPTR to a structure like this - sometimes the structure itself is meant however, so watch it.

2.5 Lists - segment lists

A third commonly-encountered BCPL structure used extensively by AmigaDOS is a "Segment list" - this is a simple singly-linked list as follows:



Again, note that this is different from the rest of the system, which tends to use the doubly-linked list structure supported by Exec - see section 1. Examples of this form of linked list are the segment list created by the scatter-loader, and the new list of "resident segments" supported in 1.2.

2.6 Resident libraries

Another mechanism used by BCPL and Tripos to deprive programmers of coffee by reducing link times (and to avoid having to duplicate commonly-used bits of code which may be needed by more than one currently-running application) is known as a "resident library". This is NOT to be confused with linker "scanned libraries" such as amiga.lib, or with Exec libraries such as intuition.library!

The AmigaDOS load-file format contains the facility to specify one or more "resident libraries" which are used by this application; these "libraries" can be any valid AmigaDOS load-files. The loader will check if the library is already resident; if not it will load it before it loads the application. It then sorts out references from the application into the resident library as part of its normal scatter-loading. In order to recognise things as calls into a resident library you need to link with an object module which includes a hunk containing resident library definitions - a special software tool is needed to create this. We're not aware of anyone making use of this facility on the Amiga, though for some applications it may be worth considering - for one thing, it has a smaller run-time overhead than the process of accessing an Exec library.

3. THE AMIGADOS KERNEL AND PROCESSES

Tripos consists of a small "kernel" written in assembler, plus various "processes" such as file handlers and CLIs, mostly written in BCPL, plus various "devices" such as "DF0:", "SER:", "CON:" etc, also mostly written in BCPL, but interfacing to "device drivers" written in assembler. AmigaDOS uses Exec as its kernel, and makes use of Exec "devices" such as trackdisk.device as its device drivers.

3.1 The AmigaDOS library

The AmigaDOS library is organised so that it can be accessed either as an Exec library (see section 2) or as a Tripos-style resident library (see 2.6 above).

Viewed as an Exec library, DOS is somewhat peculiar, in that it does NOT need to be called with the library base address in A6 (as long as you call the right address without relying on absolute memory locations somehow!), and in that it does not support Expunge(). Furthermore, its jump table doesn't actually consist of jump instructions. Instead the table consists of a series of six byte entries (same length as jump instruction) organised as follows:

```
MOVEQ     #entry_number_in_global_vector,D0
BRA       action

MOVEQ     #another_entry_number_in_global_vector,D0
BRA       action

etc.
```

While you don't need to know this to use AmigaDOS, it's quite interesting to consider what happens next. The "action" routine, which is in the library positive offset area, grabs 1500 bytes below your current stack pointer for use as a BCPL stack and sets up A1 as a BCPL stack pointer - BCPL stacks are used from the bottom upwards just to be difficult! No bounds check is done on this, so you'd better have 1500 bytes available on your stack when you call AmigaDOS, or prepare to meditate. It then zeros A0, and sets up registers A2 (= global vector), A5 (= DOS kernel action routine from ROM) and A6 (= DOS kernel cleanup routine from ROM) from locations which are also in the library positive offsets, and puts the appropriate action address from the global vector table into A4.

The ROM is now invoked by JSR (A5), and deals with the call; this often involves constructing the appropriate "packet" for the DOS call in question, sending it to the right process, and waiting for a reply indicating successful completion, or an error. Note that D1, D2, D3 and D4 are used to pass values to AmigaDOS; results are returned from the ROM action routine in D1, then transferred to D0 for return to the calling program.

3.2 Process

A Process is the basic structure used to control multi-tasking in AmigaDOS; it is built on the Exec concept of a "task", and consists of an Exec task control block, immediately followed by a message port, followed by some other bits and pieces needed by AmigaDOS. (It is quite easy to tell a Process from a Task by looking at the task control block's node type, which will be NT_PROCESS or NT_TASK respectively.) The process message port is used by AmigaDOS for its own "packet" based communication between processes. The other stuff following the message port includes the current directory lock for this process, and the current input & output "streams" - ie the handles for the current input and output files, which might very well (for example) refer to a "CON:".

Anything which calls AmigaDOS directly or indirectly (eg by opening a non-resident Amiga library or font) must be a Process rather than a simple task; an attempt to call AmigaDOS from a simple task will result in a Guru Alert since AmigaDOS will assume that it is being called from a process, will pick up nonsense values for things like current I/O streams, will try to reply to a non-existent message-port, and generally go bananas. Thus most Amiga tasks are also Processes:

"Workbench"	the WorkBench process
"CLI"	process which creates a new CLI from the Workbench (obtained from CLI icon)
"Initial CLI"	boot-up CLI with window title "AmigaDOS"
"New CLI"	a CLI obtained by CLI or NEWCLI
"Background CLI"	a CLI obtained by the RUN command
"File System"	disk file handler (one for each drive)
"CON","RAW","PRT"	handlers for appropriate AmigaDOS devices

The only simple tasks are such things as:

"trackdisk.device"	Exec device for handling disk drive
"printer.device"	Exec device for handling printer
"input.device"	Exec input device (see section 3)
"console.device"	Exec console (see section 3)

AmigaDOS refers to each process by its Process ID, which is the address (APTR not BPTR!) of the message port in the Process structure. The message port is the thing you are most likely to want to use directly, but if you need a pointer to the Process structure, you can subtract the size of a Task structure from the Process ID. Note that you can create a new process quite easily by calling CreateProc() - this takes a name, priority, segment BPTR, and stack size as parameters, and returns a new process ID. The process priority is the same as the task priority, ie a number from -128 to 127 - this is a significant difference between AmigaDOS and Tripos, since Tripos allows 65536 priorities, but does not allow two tasks to have the same priority, whereas Exec and hence AmigaDOS time-slice between tasks of the same priority.

3.3 Packet

The "packet" is the basic unit of message-passing in AmigaDOS, and is built on the Exec concept of a "message", much as a process is built on the Exec concept of a task. The way that a packet is built onto a message is by means of the "AmigaDOS Grand Bodge" (apologies to Tim King - AmigaDOS is generally remarkably bodge-free, especially given the time scale!): the part of the message structure defined as a pointer to a name is used instead to point to a Tripos structure called a "packet", as follows:

```
APTR back to message structure
APTR to reply port
LONG packet type
LONG result1
LONG result2
LONG argument1
LONG argument2
     etc.
```

AmigaDOS gets things done by sending packets like this to the message-ports of appropriate processes such as file-handlers - for a description of different packet types and functions, see the AmigaDOS technical reference manual. Note that you can also get things done by sending packets to processes, instead of calling AmigaDOS library routines - this is handy if you don't want to wait for something to finish, but want to operate asynchronously.

3.4 Co-routine

Co-routine is a term from Tripos useful in understanding AmigaDOS; the term means a routine which is executed as part of a particular process, but using its own private stack, instead of the process's standard stack.

When a co-routine is about to be invoked, memory is allocated for its stack by the process, the stack pointer is saved then altered to point at this new stack, and the new stack is initialised to contain its own size, plus a return address to the main process. Then the co-routine is invoked. When the co-routine exits, control returns to the main process code, which restores its stack pointer to the old value and deallocates the co-routine's stack memory.

This technique is used by a CLI process to execute commands - see below. The fact that the process pushes the size of the stack to the new stack is significant here, as it means that you can check if you've been given enough stack space, and refuse to run if you haven't!

3.5 CLI

A CLI (Command Line Interpreter in Tripos, changed to Command Line Interface in AmigaDOS for no obvious reason) is a Process with some special attributes, including:-

a CLI number (1 to 20)
standard input and output streams (usually a CON: device)
a path list to use when searching for command files
a prompt string definition (altered with the PROMPT command)
default stack size for co-routines for executing commands
(altered with the STACK command)

The CLI environment can be thought of as a "shell" (a term from UNIX) for AmigaDOS - it provides a user interface, performing commands typed in at the keyboard from a CON: window or read from a file, and handles input and output "re-direction", such as sending program output to the printer (PRT:) instead of the CON: window. All commands are normally loaded from disk as required, and the user can create his own commands, which have exactly the same status as those provided as part of the system.

Some special setting-up is needed to start a CLI (initialising input and output streams etc), and another process is needed to do this - it is important not to confuse the process which starts a CLI (such as the one invoked by clicking the CLI Icon on the Workbench) from the actual CLI process once running. The standard code for handling the CLI environment (as distinct from that for creating or destroying CLI processes) is in the KickStart ROM; however, under version 1.2 it is possible to load alternative CLI code, such as one which recognises resident commands, and this lives in RAM.

A CLI process can be created interactively in any of three ways:-

(1) from the CLI icon on the Workbench - this creates a new interactive CLI process with a set of default parameters, including a default stack size for co-routines of 4000 bytes. This will always be a standard CLI, using ROM code, even if a non-standard CLI has been loaded with the RESIDENT command.

(2) via the NEWCLI command from a CLI - this creates a new interactive CLI process with some parameters inherited from the CLI process which created it, including default stack size and current directory. It uses the CLI code in the resident segment list, so if this is non-standard you will get a non-standard CLI.

(3) via the RUN command from a CLI - this creates a new "background" CLI process (ie one without its own CON: window) in which to execute a specified command, whose default output stream is that of the CLI from which the RUN command was given.

A CLI process can also be created from a program, using the Execute() function in the dos.library.

The STATUS command can be used to tell you about active CLI processes. Note that these are referred to in the output from the STATUS command as Task 1, Task 2, etc - this couldn't be much more misleading, as what is actually meant is CLI 1, CLI 2, etc!

3.6 Commands

A command is code which is executed from a CLI process when it is invoked by name, and is usually loaded from a disk file with the same name as the command in the current or specified directory (in the case of a user-supplied command) or in the C: directory (in the case of a system command). In version 1.2 it is possible to specify further directory paths to be searched for commands, by using the PATH command. Also in version 1.2, by use of an appropriate CLI (not the default CLI code), commands which have been made resident using the RESIDENT command can be executed directly from memory without any disk access.

Commands are executed as co-routines of the CLI process from which they are invoked. That is, no new process is created, but stack space for the command's use is allocated and control is passed to the code at the start of the first segment of the command's code. On entry to the command's code, the registers contain:

> in A0 a pointer to the command line for execution;
> in D0 the length of the command line.

Other registers contain defined, but not documented, values; these are irrelevant to commands written in assembler or C. However, commands written in BCPL will expect to be entered with these registers correctly initialised - A1 = BCPL stack pointer, A2 = global vector etc, very much as set up by the DOS "action" routine explained in 3.1 above. This explains why it is not possible, for example, to load NewCLI or DiskEd under Wack and then execute them with GO, since these registers will not contain the appropriate values.

Commands run as co-routines under the CLI will generally start with some form of standard startup code (eg LStartup.obj or AStartup.obj for Lattice 3.03, or c.0 for Lattice 3.1), which amongst other things deals with obtaining the current CLI input and output streams by calling functions Input() and Output(), and using them to set up their own IO vectors such as C 'stdin' and 'stdout'. There is a significant difference between running from CLI and Workbench here - applications run from the Workbench run as processes rather than CLI co-routines, and have to set up default IO channels by some other method, such as opening a window on the Workbench (Lattice).

Commands are not required to preserve any registers, and the main process cannot rely on any of them being preserved when the co-routine exits.

A final point concerns "resident" commands. These are commands supported by non-standard CLIs like the Metacomco shell, which maintain a "resident segment list" of named commands in memory. Commands found in the resident segment list will be executed directly where they live in memory, without any form of loading or initialisation. While this is very fast, it means that resident commands will get in trouble if they rely on any form of data pre-initialisation, such as

```
ULONG counter = 0;
```

This will work fine the first time the command is run from the resident segment list; the next time however counter will start of with whatever value it finished with last time, resulting in general confusion. Thus to work properly as a resident command, you have to avoid this sort of thing, and use

```
ULONG counter;
counter = 0;
```

This carries a fair amount of overhead, but should probably be done if you are writing a program which can be viewed as a 'DOS extension' and which you therefore might want to make resident. Otherwise, we wouldn't bother!

.7 Re-direction

When a CLI process obtains a command line for execution (either from a CON: window or from a file) it checks for re-direction specifications, in the form of AmigaDOS file or device names preceded by '<' for input re-direction or '>' for output re-direction. The CLI will attempt to perform the requested re-direction, so that the current input and output streams are those asked for, before control is passed to the command's code. The command will only respond to re-direction if it gets its input from its default input stream (stdin in C) and sends its output to its default output stream (stdout in C). This is the reason for a problem with Wack - Wack wants to respond to single key closures, so it does its input and output to a RAW: window which it opens rather than using stdin and stdout, which means it does not support re-direction - damn it.

3.8 Devices, devices and devices

Like "library" the word device is somewhat overworked on the Amiga, being used to refer to three different things - is this an undocumented feature of AmigaSpeak? - as follows:

(1) an Exec device, as in OpenDevice(), CloseDevice(), etc, such as "trackdisk.device" or "console.device";

(2) an AmigaDOS device, which is an interface between AmigaDOS and Exec devices, and which is referred to by a name ending with a colon, which may be used in contexts where a filename is required. The standard AmigaDOS devices are

```
DF0:  handles floppy disk drive 0 file/directory access
DF1:  handles floppy disk drive 1 file/directory access
RAM:  handles the RAM disk file/directory access
PRT:  handles output to the printer
SER:  handles input/output via the serial port
PAR:  handles input/output via the parallel port
CON:  handles input/output via a CON: window
RAW:  handles input/output via a RAW: window
```

(3) an AmigaDOS logical device name, which is a name ending with a colon, may be the name of any of three types of entity:

(a) an AmigaDOS device, as in (2) above;

(b) a volume, ie a physical disk or the RAM disk, eg "C-DEVEL:" or "RAM Disk:";

(c) a directory with an logical device name set up by the system or specified via the ASSIGN command eg. "SYS:", "C:", or "THISISMEFOLKS:"

The ASSIGN command with no parameters will report on all currently known logical device names, grouping together those of the same type.

3.9 Handler processes

AmigaDOS devices (in sense (2) above) have associated processes for performing the actions required by them. These can be compared with the associated tasks used by some Exec devices such as console.device (see section 3), and are known as handler processes.

There is a function call in the DOS library to allow you to obtain the Process ID for the handler process for a named AmigaDOS device. This is DeviceProc(), for which an example call is:

```
df0handlerID = DeviceProc("DF0:");
```

Handler processes generally have the same name as their device (but without the colon), eg the handler process for a "CON:" window is called "CON". However, this is not always the case - the floppy disk handlers (ie those for DF0: and DF1:) are each called "File System".

Each open "CON:" or "RAW:" window will have its own handler process, so it is common to have several processes with these names present in the system at any one time. For this reason one cannot use DeviceProc("CON:") or DeviceProc("RAW:") to find the handler processes for these devices, since there may be more than one handler for the specified device.

Handler processes spend much of their time in wait states, waiting to receive AmigaDOS packets at their message ports asking them to perform some action, such as reading or writing data or opening or closing a file. They then perform the required action, then reply the packet to the sender.

References

For full information on disk structures, process structure, packet structures etc, see the definitions in "libraries/ dosextens.i" (don't trust dosextens.h), together with the Technical Reference and Developers sections of the AmigaDOS Manual. With any luck, you should find the latter easier going if read in conjunction with the above!

You don't really need to know BCPL to understand AmigaDOS, any more than you need to know C to understand Unix - but it probably helps. The BCPL "Kernighan & Richie" is probably "BCPL the Language and its Compiler" by Martin Richards and Colin Whitby-Strevens, published by the Cambridge University Press - all right, we admit we haven't read it! If you actually want a BCPL compiler to play with on Amiga, try screaming loudly at Metacomco.

For an "official" example of how one process can start another, see the C-program by Rob Peck which has now been widely circulated - this starts a process then sends it messages as if it had been started from Workbench. For a simpler method, which doesn't mess about pretending to be Workbench, see the example in section 1 of this book. Examples of how to talk to AmigaDOS by sending packets to processes can be found on the Fish disks.

You called,
Oh MASTER

Serial Port Debugging, and The Joy of Wack.

Illustration by Phill Legard.

Part V - Serial Port Debugging

And The Joy of Wack

A common problem encountered when developing in a multi-window environment like Amiga is - where do you put diagnostic information? There are various solutions to this:

1. You can send diagnostics to a standard channel - eg "stderr" in C - and leave it up to your compiler (or whatever) to decide what to do with it. The usual default for stderr is to use the same channel as "stdout" - thus if you are running from the CLI, errors etc will tend to go to the CLI window, and if you are running from Workbench (and linking using LStartup.obj or c.o) they will go to the window Lattice opens on the Workbench - the ability to direct diagnostics to this window is one use for this otherwise irritating feature of Lattice startup.

2. You can use a special window opened for diagnostic information only. This is one approach used by the Amiga symbolic debugger Wack, which by default will open its own RAW: window on the workbench, and converse using that.

3. You can direct debug information to an external device, such as a terminal hung on the serial port. This is the approach used by ROMWack, the resident miniature version of Wack used for emergency debugging - eg to find what is causing a guru meditation. It is also possible to run Wack on the serial port, using the -s option.

There are three main advantages to the serial port approach. First, it gets the diagnostics off the main screen, so that you can arrange things so that the Amiga screen shows exactly what the user will get while running your application, while you get diagnostics on the remote terminal. Secondly, it is reasonably proof against blow-ups - even if your screen has gone into "firework display mode" due to a graphics bug, you can still get in and find out what has gone wrong with ROMWack. Finally, the use of a remote terminal is very handy when it comes to things like printing long disassemblies or memory dumps - if you have half-way decent terminal software, you should be able to run these off to a file, then churn them out at your leisure, without involving the Amiga.

Connecting a remote terminal

The first thing to realise about debugging using the serial port is that the communication is handled, quite deliberately, in as dumb a way as possible, by directly hitting the serial port hardware. Thus you musn't expect Preferences settings or anything involving serial.device to make any difference; nor must you expect anything much in the way of clever handshakes,

XON/XOFF, RTS/CTS or whatever. The reason for this is quite simple - since you are debugging the machine it has (presumably) crashed, causing unknown memory corruption, so you certainly can't assume that things like the serial device are still alive and functioning. The only things you can reasonably assume are still there are the ROM contents and the hardware - if not you've REALLY got problems! - hence the existence of debug routines in the ROM which communicate by directly hitting the hardware.

The choice of remote terminal is up to you, though it needs to be reasonably fast - if you are going to be using Wack, then you need to be able to communicate at 9600 baud without handshake - and we recommend an eighty column screen. We use a BBC Micro running our own bi-scrolling terminal software - this allows stuff which has scrolled off the top of the screen to be "pulled back" down again which is very useful when running in trace mode (What were the register contents when I entered this subroutine?). However, this facility is not essential, so pretty well anything will do - you could probably use a Commodore 64 with appropriate software and a serial cartridge, or, of course, another Amiga.

Since there is nothing fancy to worry about in the way of handshake conventions etc, connecting up a cable is quite simple, the only trick being that you may have to do something with the handshake lines (RTS/CTS and/or DSR/DTR) at the terminal end if it is expecting a handshake. The connections for a BBC Micro are as follows:

Amiga Transmit data (pin 2) to BBC data in (pin A)
Amiga Received data (pin 3) to BBC data out (pin B)
Amiga Signal ground (pin 7) to BBC 0V (pin C)

BBC CTS (pin D) jumpered to BBC RTS (pin E)

For other terminals it may also be necessary to jumper DSR to DTR - as is nearly always the case with horrible serial comms, you'll just have to try it and see.

The terminal software should then be configured as follows:

9600 baud send
9600 baud receive
8 data bits, 1 stop bit, no parity
linefeeds expected (ie don't auto-linefeed after return)
all fancy protocol options off

To see if everything is working, open a CLI in the Amiga then invoke ROMWack - this should result in a display on the remote terminal similar to the one shown in figure 1. If not, check baud-rate settings at the terminal end and the three connections between the Amiga and the terminal. Now press RETURN on the remote terminal - this should cause a one line "frame" showing memory contents to be displayed as shown in figure 2. If nothing happens, the terminal isn't sending - check the Amiga received

data connection, then try different jumpering between CTS/RTS and DSR/DTR until something happens. Don't you just hate serial communications?

Once you have got this working, you can explore ROMWack as explained below; alternatively if you just want to exit ROMWack then you can get out of it and bring the Amiga back to life by typing "resume". Note that ROMWack completely kills all normal system activities and multi-tasking, so you won't even be able to move the pointer using the mouse (and the disk light won't go off) until you do this.

Undocumented routines in Exec

All the Amiga's serial port debugging facilities make use of some originally undocumented routines in the Exec library; these can be used directly by making use of the interface code in the special linker library debug.lib, and are also used internally by ROMWack, Wack, and Exec's Alert system. The reason for these routines being originally undocumented was (presumably) because they might not still be there in later releases of the Amiga as space in the ROM got tighter and tighter - however we note that in the 1.2 autodocs RawDoFmt() at least has made it as an officially documented system facility! Anyway, if the purpose is simply to debug an application on the current version of the software, then it certainly doesn't hurt to know about these routines, so here they are:

offset	name	description
FDF6	RawDoFmt	format data into a character stream
FDFC	RawPutChar	put character to debug console
FE02	RawMayGetChar	get char from debug-console if there's one pending
FE08	RawIOInit	initialise for I/O from debug-console

A summary of these functions is as follows:

RawDoFmt - Exec library call invoked by debug.lib routine KDoFmt - output a stream of characters formatted in a way similar to C printf(). Register usage is as follows:

- A0 - pointer to null terminated format string, eg 'Longword value = $%lx',10,0
- A1 - pointer to data-stream - eg a pointer to a long-word value $21222324.
- A2 - PutChProc - address of routine to use to output a single character - eg interface to RawPutChar
- A3 - PutChData - anything extra you want to pass to PutChProc.

When this routine is called, the PutChProc routine specified will be called repeatedly with the next character to output in the bottom byte of D0, and whatever you put in A3 still in A3 (up to

you). Given the example format string and data-stream given above, the output would be "Longword value = $21222324", followed by a linefeed then a terminating null.

RawPutChar - Exec library call invoked by debug.lib routine KPutChar - puts a character to the "debug console" by slinging it directly at the serial port hardware - called with character in D0.

RawMayGetChar - Exec library call invoked by debug.lib routine KMayGetChar - gets character from serial port - returns char in D0, or -1 if no character waiting.

RawIOInit - initialisation routine used by Wack and ROMWack to set the serial port hardware to 9600 baud send/receive, 8 data bits, 1 stop bit, no parity bit. Not available from debug.lib for some reason.

More about DoFmt

The two principal inputs to RawDoFmt (and routines which use it such as KPutStr() discussed below) are a pointer to a format string in A0 and a pointer to a data stream in A1.

The format string follows C conventions as regards format instructions (%lx etc), but can also contain explicit codes for linefeeds etc, so in assembler you might have

```
MyFormatString dc.b 'The answer is $%lx',10,0
```

to output a single value as longword hexadecimal, followed by a linefeed, followed by the usual null string terminator. Other valid format specifications are as follows:

```
%ld  -   longword value printed in decimal
%lx  -   longword value printed in hex
%lc  -   least sig byte of longword printed as a character
%lo  -   longword value printed in octal
%s   -   successive string chars printed up to null terminator
```

It is also possible to specify field widths as in

```
%5ld -   longword value in decimal in 5 char field, right
         justified.
%06lx-   longword value in hex in 6 char field, right
         justified, padded with leading zeros.
```

The "data stream" input to DoFmt simply consists of a pointer to the data to be printed, stored as a succession of longword values and/or null-terminated strings. A convenient way of setting this up is to use a "print stack" using one of the address registers as a stack pointer - successive values to be output using DoFmt can be pushed to this stack by instructions like

```
move.l    nextvalue,-(a1)
```

This will end up with a1 pointing to start of the data stream (with the last value pushed as the first value to be output), ready to call DoFmt.

Using debug.lib

Debug.lib is a linker scanned library containing interface code to enable the routines above to be called conveniently from assembler or C, plus some additional routines built on top of them. This allows you to output diagnostic messages of the sort "Entering routine DeepThought()", "Current value of Meaning_Of_Life is 41" etc to the serial port while you test your application. This can be massively useful, particularly in sorting out high level bugs involving control flow. It is good practice to make use of conditional assembly or compilation to include or exclude calls of this nature, depending on whether you are creating a development or production version of your software.

A point to note about debug.lib is that since it makes direct use of the serial port hardware, and since you won't be using RawIOInit (unless you write your own interface code to call the Exec library directly), the communication will take place at whatever rate the hardware is currently set to. The default power-up is 9600 baud, 8 data-bits etc as used by ROMWack, and this is not affected by the current setting of Preferences. However, if you change Preferences to (say) 1200 baud, and then actually output something at this baud rate (eg ECHO >SER: "ECCE WOMBAT"), then the hardware settings will be changed to 1200 baud; subsequent calls via the debug library will therefore operate at this baud rate. It is also possible to change the number of data bits and stop bits by this method; it is not however possible to enable a protocol such as CTS/RTS, as this depends on software in serial.device, not just hardware settings. Changing the baud rate used by debug.lib can be useful - eg when running a utility like SNOOP which outputs a lot of data to a terminal which has trouble keeping up with 9600 baud.

Routines in debug.lib

There are three types of routine in debug.lib:

1. Interface to undocumented calls in Exec. These are KPutChar, KMayGetChar and KDoFmt - see above.

2. Routines built on undocumented calls in Exec. These are as follows:

   KGetChar - keeps calling KMayGetChar until it actually gets something. Returns with character in D0.

KGetNum - input a number from the debug console, with echo. Allows a signed decimal integer to be input on the remote terminal, terminated by carriage return. Returns with the number in D0. Uses KGetChar and KPutChar.

KPutFmt - put formatted data to debug console. Called with A0 pointing to format string, A1 pointing to values to be output according to format string. Sets A2 to address of KPutChar, then calls KDoFmt.

KPutStr - put null terminated string to debug console. Called with address of string in A0. Uses KPutChar.

3. Stand-alone routines. There is currently only one of these, which is KCmpStr, to compare two null terminated strings. Called with string pointers in A0 and A1, returns with D0 = 0 if strings equal, or D0 = char where strings differ or where one string terminates - negative value indicates second string is longer.

The debug library is quite interesting from another angle, as it provides a short and straightforward example of how a linker scanned library is constructed by concatenating object modules, of how an Exec library is called at assembler level, and of how a function written in assembler receives paramaters from a calling program written in C by reading them off the stack. It is possible to obtain a listing of it using a disassembler like ones to be found on the Fish disks, or as part of the "Metacomco toolkit" - this is a recommended exercise.

Other debugging tools - SNOOP

Besides putting diagnostics directly in your code using KPutStr, KPutFmt etc, it is also possible to run diagnostics in other processes that keep an eye on what's going on; things like stack-watchers can be implemented like this, which can be quite useful. Note also that since the diagnostic routines make no use of AmigaDOS, there is no problem in calling them from tasks which aren't processes, such as sub-tasks spawned by your application.

A good example of a debugging tool which uses debug.lib is the SNOOP utility, supposedly provided (with source code) on the 1.2 toolkit disk. This uses Exec routine SetFunction() to replace the standard system AllocMem() and FreeMem() by "snooped" versions which output a diagnostic message to the serial port on every memory allocation and deallocation - this is very useful in tracking memory leaks.

Note that you may get flooded with information from SNOOP, since you will get a message on _every_ memory allocation/deallocation, including ones made by the system - if you run SNOOP then try (say) resizing a window on the Workbench, you will notice that there are an awful lot of these! This can cause two problems:

1. Your remote terminal may have trouble keeping up at 9600 baud without handshake. If so, try running at a lower baud rate - see under "Using debug.lib" above for how to do this.

2. You may have trouble disentangling your memory allocations/deallocations from the ones made by the system. One solution to this is to pass all memory allocations through your own routines (we use something called the Ariadne Manager which also tracks what we're up to), and put diagnostics in this using KPutFmt etc. Another option is to modify SNOOP so that it checks which task is doing the AllocMem or FreeMem, and only outputs diagnostics if it is (one of) your task(s).

The joy of Wack

By making appropriate use of "higher-level" debugging tools like the stuff in debug.lib, you will probably be able to avoid having to go in at a lower level most of the time - on a machine as complex as Amiga, we would heartily recommend this! However, if you really have to, or if you just like messing about with symbolic debuggers, then there's always Wack.

There has been some debate over the origin of the name "Wack". Bill Donald's suggestion that it stands for "Westchester Amiga Crash Killer" can sadly be discounted, since the documentation with 1.2 states that "The name derives from its ability to wack bugs over the head". Pity.

There are two main versions of Wack - the full disk-loaded symbolic debugger known as GrandWack, and a cut down "ROM"-resident version for emergency debugging known as ROMWack. GrandWack can talk either via the serial port, or via its own RAW window on Workbench; ROMWack is serial port only.

There have (of course) been various revisions of GrandWack. The original program (Wack 1.0) put out with versions 1.0 and 1.1 of the ROM kernal contained a core of general purpose debug facilities, plus a number of Amiga-specific commands such as "devices", "libraries", "resources" etc; however there was no simple way of adding new facilities. The new version of Wack put out with version 1.2 (the latest we have is 1.004) is about twice as long as the old one (presumably we all have fast RAM expansion these days!), has lost most of the old Amiga-specific commands, but has gained a very powerful general-purpose macro facility, with a LISP-like syntax! Using this, it is possible to recreate the old Amiga-specific commands (Wack now looks for a default macro-set s:wack-macros which hopefully will contain these); it is also possible quite easily to modify existing commands, or to add new ones for special purposes. We shall refer to the two versions as OldWack and NewWack when it comes to pointing out the differences.

ROMWack

ROMWack is the Amiga's equivalent of a "machine code monitor", roughly comparable with the SYS 4 monitor in the old PETs; GrandWack can in this respect be compared with something more like Supermon. Again, for reasons of ROM-space, the ROMWack facilities have been kept fairly rudimentary, and most notably absent are any ability to work in terms of symbols rather than absolute addresses, and any form of disassembler. This makes finding your way around with ROMWack pretty slow going. ROMWack is best viewed as an emergency measure, useful for pinning down things like the causes of guru meditations when other measures (like thinking about it) have proved unsuccessful. ROMWack is best used in conjunction with more powerful tools such as Wack - eg if you have loaded using Wack and found out where everything of interest is, it is going to make life much easier if you need to pin down the cause of any serious problem using ROMWack.

ROMWack does its best to disturb the machine as little as possible, and to ensure that any process that has gone out of control does no further damage. To this end, it disables multi-tasking (while keeping interrupts enabled) - the documentation warns you that this may lead to buffer-full problems if you spend too long using ROMWack (too many key closures etc), so you should aim to find out whatever you need, then get out fast. For workspace, ROMWack uses a small amount of supervisor stack, and memory between $200 and $400 - these are normally reserved for 68000 "User vectors" for a form of clever exception (interrupt) processing which is not used on the Amiga, since Amiga prefers to do its own clever tricks with interrupts using PAULA.

Entering ROMWack

ROMWack is normally entered using the Debug(0) function in the Exec library, though note that this function can be arranged to invoke GrandWack or some other debugger - see the -r option in GrandWack discussed below. Calls to Debug(0) can be put in your code for diagnostic purposes to put you into ROMWack (or Wack or whatever) in circumstances where you want to have a peer around in memory before proceeding, or if you want to single-step - the entry PC value shows where Debug() was called from and the stack frame SF: shows what is currently at the top of the stack - ie the next few return addresses, so you should be able to find out where things are without too much difficulty. Typing "resume" allows you to leave ROMWack and to pick up your application again after the call to Debug().

ROMWack can also be invoked directly from the CLI using the program ROMWack - this must be about the shortest utility available on the Amiga, since it currently goes

```
rom-wack
PC: FC08B8  SR: 0000  USP: 2118D0  SSP: 07FFFA  XCPT: 0000  TASK: 20F488
DR: 0000002D 0000002D 00000FA0 00000FA8 00000001 0000003E 000840C5 0020F4E4
AR: 00000000 00211200 002033E0 002105C4 0020F488 00FF44B4 00000676
SF: 0020 E11C 0000 0676 0020 97B2 0020 9AA0 0021 0698 0000 0001 0021 07C4 00FF
```

Fig 1 - ROMWack entry via Debug(0)

```
om-wack
C: FC08B8  SR: 0000  USP: 2118D0  SSP: 07FFFA  XCPT: 0000  TASK: 20F488
R: 0000002D 0000002D 00000FA0 00000FA8 00000001 0000003E 000840C5 0020F4E4
R: 00000000 00211200 002033E0 002105C4 0020F488 00FF44B4 00000676
F: 0020 E11C 0000 0676 0020 97B2 0020 9AA0 0021 0698 0000 0001 0021 07C4 00FF

C08B8 4E75 007C 2000 518F 40D7 2F7C 00FC 08B8  N u.. !  .. Q.. @.. / !....^H..
```

Fig 2 - 'frame' obtained by pressing return

```
om-wack
C: 203A5A  SR: 0000  USP: 2180A8  SSP: 07FFFA  XCPT: 0021  TASK: 208B18
R: 00000001 00208E60 00003A98 00003AA0 00000001 0000003E 00080E21 00208B74
R: 00208E60 00208F6C 002033E0 00203A4C 002180B0 00FF44B4 00FF44A8
F: 0020 3A56 00FF 4CAE 0000 3A98 0020 91F4 0000 0000 001E 7F48 220B E489 7020
```

Fig 3 - ROMWack entry via Guru Meditation

```
om-wack
C: 203A5A  SR: 0000  USP: 2180A8  SSP: 07FFFA  XCPT: 0021  TASK: 208B18
R: 00000001 00208E60 00003A98 00003AA0 00000001 0000003E 00080E21 00208B74
R: 00208E60 00208F6C 002033E0 00203A4C 002180B0 00FF44B4 00FF44A8
F: 0020 3A56 00FF 4CAE 0000 3A98 0020 91F4 0000 0000 001E 7F48 220B E489 7020

03A5A 0000 0008 18C3 0020 3B58 0000 0038 0020 ......^H^X....   ; X...... 8..
```

Fig 4 - 'frame' obtained by pressing return

```
m-wack
: 203A5A  SR: 0000  USP: 2180A8  SSP: 07FFFA  XCPT: 0021  TASK: 208B18
: 00000001 00208E60 00003A98 00003AA0 00000001 0000003E 00080E21 00208B74
: 00208E60 00208F6C 002033E0 00203A4C 002180B0 00FF44B4 00FF44A8
: 0020 3A56 00FF 4CAE 0000 3A98 0020 91F4 0000 0000 001E 7F48 220B E489 7020

3A5A 0000 0008 18C3 0020 3B58 0000 0038 0020 ......^H^X....   ; X...... 8..

3A4A 0014 0000 0000 4EB9 0020 3A58 4E75 4E41 ..^T........ N....   : X N u N A
```

Fig 5 - previous frame obtained by pressing ','

```
main()
{
   printf("Entering ROM-Wack on serial port (9600 baud)\n");
   Debug();
}
```

Note that this two-line program contains two errors (!), first because it won't necessarily invoke ROMWack at all if something else (GrandWack) has been made "resident", and second because it calls Debug() with no parameters, instead of a single null parameter, as recommended.

Finally, ROMWack can be entered following a dead-end Alert (Guru Meditation), to allow you to poke about to find out what went wrong, before resetting the Amiga. In version 1.1, you wait until you have the (familiar?) message up

"Software Failure. Press left mouse button to continue"

then press the right mouse button - this will put you into ROMWack. In 1.2, the mechanism has changed. Right mouse button on Guru Meditation no longer works; instead the system watches the serial port data lines for a pattern of "seven ones", while it is flashing the light before actually displaying the guru. Thus pressing BREAK or DEL ($7F) on the remote terminal while the light is flashing will put you into ROMWack. In either of these cases, exiting ROMWack by typing "resume" will cause the Amiga to reset.

ROMWack example

As mentioned above, debugging with ROMWack is usually a last resort, and should normally only be tackled when you have already found out where everything is using Wack or another symbolic debugger, and even then it takes ages! The "simple example" which follows is therefore, we freely admit, a total fake, but may help illustrate the principles of serial port Wackery.

In this example, the Amiga has come to a halt due to an unrecognised 68000 exception (Motorola terminology meaning "soft interrupt" cf 6502 BRK - not to be confused by what Amiga mean by "exception" or "software interrupt" - see section 1.) The symptoms of this are that the system first of all puts up a requestor:

```
 ------------------------------
| Software error - task held   |
| Finish ALL disk activity     |
| Select CANCEL to reset/debug |
 ------------------------------
```

You should now go around cleaning up anything else you may have running at the time, saving everything that may be necessary to disk. When you have done this, selecting CANCEL in the "Software

error" requestor brings up the familiar "alert" (pushing down other screen contents in this case):

```
------------------------------------------------------------
|     Software Failure.   Press left mouse button to continue.  |
|             Guru Meditation #00000021.00208B18                |
------------------------------------------------------------
```

This indicates that we have taken exception vector 33 ($21), which corresponds to 68000 TRAP #1, and that no trap-handling has been set-up to allow the task in question to know what to do about it. The 68000 supports 16 TRAP instructions all of which function a bit like 6502 BRK, causing it to go into exception processing (interrupt) mode, saving off status register, program counter etc, entering supervisor mode, then continuing execution from an address held in one of 16 vectors (vectors 32 to 47). An Amiga task can be set up to handle traps (consisting of actual TRAP instructions and other conditions causing processor exceptions such as address errors, illegal instructions, or divide by zero) by setting a vector tc_TrapCode in the task control block - if this has not been done, then the system will assume something has gone wrong, and will perform its default trap handling, which is to give the "Software error" requestor followed by the "Software failure" alert.

Note that this form of Guru Meditation, in which the Meditation number starts with four leading zeros, always results from a processor trap, and is the most common form of Amiga failure. Another form of Guru Meditation, in which the first part of the meditation number is non-zero, can arise from a variety of different error conditions in different libraries - for more information on this, see section 1 of this book, and header file exec/alerts.h.

The second part of the guru meditation number - after the dot - indicates that the task control block for the task which has gone guru is at address $208B18 (in fast memory). This is all we can find out without using ROMWack.

Assuming we have a 9600 baud remote terminal connected as described above, we can invoke ROMWack by pressing right mouse button on the Guru (ROM version 1.1), or pressing remote BREAK or DEL before the Guru (version 1.2). This gives a display on the remote terminal as shown in figure 3. The top line shows that our program counter was at $203A5A when we got into trouble; also displayed are our current status register, user stack pointer, supervisor stack pointer, exception number (as shown in the guru alert), and the address of the control block for this task (also as shown in the alert). Below this are displayed the current contents of data registers D0 to D7, followed by the contents of address registers A0 to A6 (A7 is the user stack pointer, and has already been displayed.)

Below the registers is displayed a "stack frame", showing the fifteen words on the user stack immediately below the current

setting of the user stack pointer. From this we see that the routine that went guru was going to return to the routine that called it at $00203A56 (just a few bytes earlier), and that this routine was going to return into the ROM at 00FF4CAE. Below this we can see the current value of A3, which was obviously pushed by this ROM routine.

If we now press <RETURN> on the remote terminal, we get a "frame" of 8 words starting from the program counter value where we crashed - this doesn't tell us very much (fig 4). Pressing "," now takes us back a frame (fig 5) - here we see the TRAP #1 instruction ($4E41) that got us into trouble at $203A58 (the PC was incremented by a word after fetching this instruction and before executing it, which is why we entered ROMWack with PC = 203A5A). Before this we see an RTS ($4E75) at $203A56 - this is the address we were going to return to if we hadn't hit the trap - and before this we see the JSR $203A58 which pushed this return address (4EB9 0020 3A58). With a bit (or a lot) more sweat we should now be able to investigate further, and find out why this happened.

In fact, as mentioned above, this example was a fake, and the alert was caused by running a program "guru" from the CLI which was assembled from source code guru.asm as follows:

```
     section   guru
     jsr       crash
     rts

crash:
     trap      #1
     end
```

This was assembled by itself then linked without any libraries or any form of startup. The code ended up being scatter loaded to $203A50, with subroutine "crash" at $203A58. The ROM return address $FF4CAE would be part of the AmigaDOS code concerned with executing commands typed at a CLI - however, note that for a lot of its activities AmigaDOS uses its own funny upwards-growing BCPL stack using A1 as a BCPL stack-pointer, so we would have to look at this stack as well if we wanted to trace the AmigaDOS part of the story!

ROMWack facilities

ROMWack is documented in ROM Kernal manual Volume 1. A summary of the facilities offered (updated to 1.2) is as follows:

Current address

ROMWack works in terms of a "current address", initialised to the value of the program counter on entry. The current address can be changed by typing in a hex number terminated by <RETURN>. Generally, BACKSPACE and CTRL-X function as usual when entering

numbers or symbols into Wack.

<u>Single key facilities</u>

Most ROMWack facilities are "permanently bound" to particular single keys. These are as follows:

Key	Facility
?	simple help - display list of symbols
return	display current frame
.	forward to next frame
,	backward to previous frame
space or >	forward by one word
bs or <	backward by one word
+n	forward by n bytes
-n	backward by n bytes
:n	set frame size (number of bytes displayed on line) to n. Default is 16.
[	indirect - set new current address to contents of old current address; remember old current address on an "indirection stack". For example, entering "4" will take you to AbsExecBase; typing "[" will now take you from there to SysBase. The Exec library structure at this location starts with a node which starts with a pointer to the next node in the system's library list - typing "[" again takes us to this library. Typing "+a" moves us ten bytes on to the pointer to the name of this library; pressing "[" once more takes us to this name - probably "expansion.library".
]	exdirect - go back one level up the indirection stack. For example, if you have just followed the indirection example given above, then pressing "]" 3 times will take you back to AbsExecBase (4). Be warned that in some versions of ROMWack, typing "]" again at this point - ie attempting to exdirect with nothing on the indirection stack - can be fatal and result in another guru alert!
=	alter memory word. Prompts with old value, then allows you to enter new value.
!	alter register value. Registers which can be altered are A0 to A6, D0 to D7 and U (user stack pointer) - eg "!A0". Prompts with old value, then allows you to enter new value.
tab	single step - execute next instruction (from current setting of PC) and redisplay register values etc. Useful for single stepping having interrupted execution using Debug(0).

Functions invoked by symbols

Other ROMWack facilities are invoked by typing a "symbol" (such as a command like "resume") and pressing return - in ROMWack symbols are permanently bound to particular actions. Some symbols also have single key alternatives.

Block memory operations such as "fill" or "find" go from the current address up to another address set by "limit" or "^". For example, to fill memory from 300000 to 300080 (exclusive) with a specified pattern, first set the current address to 300080, then type "limit", then set the current address to 300000 and type "fill" - the system will them prompt you for a fill pattern.

ROMWack symbols are as follows:

regs	shows current "register frame", as displayed on entry to ROMWack.
alter	alters memory by giving repeated prompts as for "=". Exited by null input (return).
^ limit	set limit at current address
find	prompts for a pattern of up to four bytes, then searches memory from current address up to limit, stopping with current address at first occurrence of the pattern, or at limit if pattern not found.
fill	prompts for a pattern of up to four bytes then fills memory from current address up to but not including limit.
go	continue execution from current address.
^D resume	continue execution from PC address - ie from after where you entered ROMWack, or the next instruction after the last one you executed while single stepping.
ig boot	re-boot the Amiga. ig???
set	set breakpoint at current address - saves word at current address, then inserts a TRAP #15 (opcode $4E4F causing exception $2F). If you now continue execution by "resume" (or "go" from an appropriate current address) then hitting the breakpoint will put you back in ROMWack; ROMWack will recognise this as a breakpoint entry, and will remove the trap and restore previous memory contents, so that "resume" will allow execution to continue. You can set up to 16 breakpoints.
clear	clear breakpoint at current address - ignored if current address is not a breakpoint!
show	display addresses of current breakpoints.
reset	get rid of all breakpoints.
user	forces the Amiga back into multi-tasking after a crash, hopefully to allow disk buffers to be written out before you reset. Leaves the problem that caused the crash unresolved - ie the task that went guru will still have memory, resources or whatever allocated to

it, so you ought to reset anyway. Note that "user" may not work at all, eg if the crash in question has totally corrupted memory!

list
: attempts to display an Amiga "list" starting from a node at the current address, and displaying node addresses, types, priorities and name. Seemed a bit bugged - at least in early versions of 1.2.

GrandWack

GrandWack is the full disk-loaded Amiga symbolic debugger, and contains a number of features that make life a lot easier than when using ROMWack. Most significant of these are the presence of a 68000 disassembler, a facility to work in terms of meaningful text symbols instead of directly in machine addresses (hence "symbolic debugger"), a number of useful standard commands relating to Amiga libraries devices etc (OldWack), and a facility to define your own commands (NewWack).

GrandWack symbols

Symbols in GrandWack fall into the following categories:

Primitive symbols or commands are the commands understood by Wack when loaded, similar to the symbols understood by ROMWack. In GrandWack however, ALL actions have symbols associated with them, and these symbols may or may not have keys "bound" to them; thus the key "?" for example is bound to the symbol "help", which in turn invokes the help function.

Offsets and bases are symbols associated with locations in the program currently being debugged. Amiga load files consists of a series of "hunks" which in turn contain various block-types, including "hunk_symbol" blocks containing symbolic information, associating text labels with particular offsets in the hunk in question (see section 2); this information is normally ignored by the AmigaDOS scatter-loader, but it is available for use by symbolic debuggers like Wack. The standard scanned libraries like amiga.lib already contain symbolic information; if you are using Lattice C, you can instruct the compiler to output symbolic information about your own labels by using the -d option. You can then instruct Wack to load your program and the symbolic information by

```
Wack <progname> [<arguments>]
```

This invokes Wack, and causes it to scatter-load <progname> and pass <arguments> to it exactly as if they had been entered after the program name from the CLI. As Wack loads, it takes a note of the hunk base addresses and associates them with symbols `hunk_0, `hunk_1 etc (~hunk_0 etc in OldWack); it also notes the actual addresses corresponding to the offsets associated with the

symbols defined within the hunks. GrandWack allows you to use a symbol like this whenever you might otherwise use an absolute address; thus typing

    `hunk_0

(or ~hunk_0 in OldWack) will set the current address to the start of the first hunk in the program, and typing

    _Thing

will take you to a label _Thing (if defined), which might correspond to the start of a C function Thing() compiled using the -d option, the leading underscore being tacked on by Lattice. If you haven't used a symbolic debugger before, you won't believe how liberating this is until you try it!

> WARNING - Lattice 3.1 and later introduce some new hunk types, which old versions of Wack (and Alink) won't recognise - this causes Wack to fail with 'unrecognised hunk type'. Hopefully, this will be fixed when NewWack is finally officially released.

Registers are symbols associated with particular processor registers. Examples are !d0 to !d7 and !a0 to !a6 as in ROMWack, plus !pc and !sp for program counter and user stack pointer.

Macros and variables are symbols associated with user-defined commands (NewWack only), and with variables used within those commands, just like in any interpreted language.

Data types

The data types supported by NewWack are as follows:

Numbers - the default for these is hex, as in ROMWack; however it is possible to use decimal if you want to, by using a leading "#".

Characters - prefaced by a single quote, eg 'A. Control characters can be represented using a leading ^ - eg ^C.

Strings - enclosed in double quotes. All C-type escape sequences are allowed - \n for newline, \t for tab, \b for backspace, \0 for null, \\ for backslash, \' for single quote, \" for double quote, and \ooo for byte with octal value ooo. In addition \(and \) must be used for open and close parenthesis, in order not to confuse NewWack's LISP-like parser.

Key bindings

As you might expect, Wack maintains linked lists associating its symbols with the appropriate addresses; it also maintains lists associating individual keys with particular functions, so that hitting the key in question is equivalent to typing in the associated symbol. In fact, in OldWack, all keys are bound to functions, keys like the alphanumerics, delete and cancel being bound to functions ~GatherKeys, ~GatherDelKey, ~GatherCancelKey which deal with inputting a symbol, and processing it when return is pressed. NewWack has the facility to make your own bindings by defining your own "key macros".

Wack options

Besides the facility to follow Wack with a filename (and optionally with the rest of a CLI command line) for symbolic debugging, the following options can also be specified:

- -s run GrandWack down serial port, like ROMWack. Note that OldWack used to mess up disassembler output to the serial port, and also have trouble with "quit" if run with this option.
- -r make GrandWack resident so that any Debug(0) calls within the program being debugged drop you into GrandWack instead of ROMWack. Note that this has nothing to do with being "resident" in the sense of AmigaDOS's new "resident segment list" (see section 4). Note also that if a "resident" Wack 1.004 is being entered using Debug(0), it assumes that it is being called from the same process as it was originally invoked from, and gets upset if it isn't. Thus setting up Wack -r from one CLI, then calling Debug(0) by entering ROMWack in another CLI results in Wack getting very confused about where its input is supposed to be coming from - not recommended.
- -c specifies a command file from which NewWack is to fetch macro definitions, and key-bindings. The default is to use s:wack-macros.

OldWack facilities

A summary of OldWack 1.0 can be obtained by typing "?" or "help"; a full list of key bindings can be obtained by typing "keys", and a full list of symbols can be obtained by typing "symbols" (use right mouse button to stop them scrolling off the screen before you can read them).

```
?    help
     keys
     symbols
```

Facilities the same as ROMWack

OldWack supports most of the single key commands understood by ROMWack, now bound to appropriate symbols; it also supports most of ROMWack's symbols, but without "limit", "find" and "fill". In a few cases key bindings are different - eg "!" is bound to "resume" while ^D is bound to "~exit" - and in other cases the exact effects of the commands are slightly different - eg the format of the register frame produced by "regs" isn't the same in GrandWack as in ROMWack. Facilities more-or-less the same as ROMWack are as follows:

ret	^J	show_frame
	.	next_frame
	,	back_frame
spc	>	next_word
^H	<	back_word
	+n	next_count
	-n	back_count
	:n	size_frame
	[	indirect
	]	exdirect
	=	assign_mem
	^	set
		clear
		reset
		show
		go
	!	resume
	^D	~exit
	`	regs
tab	^I	step

Disassembler

The GrandWack disassembler is invoked by ";" which is bound to symbol "disassemble"; this will disassemble the current frame (number of lines depends on frame size) and increment the current address accordingly. An alternative disassembler is invoked by "/", which is bound to "newdisasm".

;	disassemble
/	newdisasm

Commands which display system lists

The following commands give formatted displays of various lists maintained by the system:

devs, devices	-	all currently resident devices
libs, libraries	-	all currently resident libraries
rsrcs, resources	-	all resident resources
ports	-	all public message ports
mods, modules	-	all resident modules
regions	-	all regions of memory in system
mem, memory	-	free memory blocks
tasks	-	all tasks running, ready, or waiting

Note that OldWack "tasks" first looks down the task ready queue, then the task waiting list; this means that tasks like input.device which will be moving between the two lists while Wack is running may appear once, twice, or not at all!

Commands which format from current address as a particular system structure

To use the following commands, you must first set Wack's current address to the start of the structure in question:

showtask	-	give details of task control block at current address.
showprocess		give details of process structure (task control block plus other stuff) at current address.
listsegs	-	list segment chain starting at current address.

Miscellaneous

Other commands supported by OldWack are as follows:

execbase	-	give details of ExecBase structure
ints	-	list details of interrupt handling
seglist	-	list all segments in Wack-loaded program
@ where	-	gives current address as an offset relative to the last symbol in Wack-loaded program, else "you're lost".
quit	-	return to CLI.
magic,xyzzy		guess!

New Wack for Old

A full discussion of all the facilities available in NewWack is beyond the scope of this section; it is also, thankfully, unnecessary, as NewWack comes with comprehensive documentation - at last! However, a summary of changes and new facilities is as follows.

Lots of Irritating Spurious Parentheses

NewWack has lost most of the built-in Amiga-specific commands listed above, but has gained a very powerful general-purpose "build your own commands" facility, using a LISP-like syntax. This means that if you want NewWack to actually output the value of something, you have to put it in parentheses. NewWack supports a wide variety of arithmetic operations, using a LISP-like pre-fix notation, so

```
+ 2 2
```

wil be taken as a command to NewWack to add 2 and 2, but not tell you what the answer is (useful). To get an output, you have to enter

```
(+ 2 2)
```

which causes NewWack to output

```
Value: 4
```

Similarly

```
allocmemory 1000
```

will allocate a 4096 byte buffer, but not tell you where it is, so that there is no way either of using it or of getting rid of it (*really* useful!); the sensible thing to do is

```
(allocmemory 1000)
```

which will cause NewWack to output a pointer, eg

```
Value: 2C2F0
```

NewWack capabilities

An very rough outline of NewWack facilities follows - for detailed information see wack.doc.

Frame control, etc This is very much the same as in OldWack, though in some cases the commands have been enhanced - eg "disassemble" now labels each line with symbolic information, and is more sensible about multi-register instructions.

Arithmetic, logic & comparison NewWack supports a wide range of arithmetic, logical and comparison operations - the latter are mainly used in macros.

Memory access NewWack allows memory to be altered using "assign_mem", bound to the "=" key, just as in ROMWack and OldWack. NewWack also allows you to update locations other than the current address, and supports a "peek" operator, allowing the contents of memory locations to be used in expressions. NewWack also supports "find" and "fill" but with a different syntax to ROMWack. Finally, NewWack supports a "copy" instruction.

Breakpoints NewWack supports breakpoint set, clear, reset and show but with a different syntax to OldWack; it also supports new commands "halt" and "is_break".

Single-stepping Single-stepping is accomplished as before with the "step" command, though this now takes an optional parameter allowing you to execute more than one instruction before returning to Wack. To return to normal execution after single-stepping, use "go" - note that in Wack 1.004 there is no "resume" command, and that "go" continues from the current PC value, not from the current address! Finally, to skip a subroutine call while single-stepping, use "stepover". This sets a breakpoint immediately after the next instruction, then executes "go".

Input/output The only output command in NewWack is "print" - this outputs values according to a data stream like C printf() or debug library KDoFmt. NewWack can load programs and macros when first invoked; it also supports commands "load" to load new programs or macros, and "bindsymbols" to fetch symbol definitions from a file without actually loading it.

Macros The command used by NewWack to define a user-defined command is "macro <name> <expression>". If expression starts with an open-parenthesis, it can stretch over many lines, and will only terminate on a matching close-parenthesis - this allows very complex expressions incorporating logical tests etc to be built up quite easily.

Key Macros In much the same way as it lets you define your own commands, GrandWack lets you make your own key bindings known as "key macros".

Control flow A number of commands allow conditional execution within macros, the simplest of which is "if", which is a full if...then...else. Slightly more complex is "select", which permits a series of conditions, each with associated "action expression", to be evaluated, until one evaluates TRUE. Looping is possible using "while"; "for" is also supported, and is treated, as in C, as a special case of "while".

System interaction As mentioned earlier, most Amiga specific commands have been removed from NewWack, as they can be implemented using macros. However, a few system interaction

commands are needed, particularly to let you do things like disabling task switching while examining system lists - these are "permit", "forbid", "enable", "disable", "allocmemory" and "freememory".

Internal data structure access Finally, within macros it is sometimes useful to examine NewWack's internal data strucures directly, working in terms of pointers to the nodes in Wack's internal linked lists. A number of commands permit you to do this.

References

ROMWack is documented in ROM Kernel Manual Volume 1 - though note that it has grown some new facilities for version 1.2.

debug.lib is documented in ROM Kernel Manual Volume 2; an update to this should be available with 1.2.

Source code for SNOOP should be provided with the 1.2 toolkit - this is both a useful utility in itself, and a good example of using debug.lib.

"NewWack" documentation should be available as wack.doc on the 1.2 toolkit.

Section 6 - Introducing Amiga Graphics

Illustration by Shelley O'Neil.

Section 6 - Introducing Amiga Graphics

Trying to write this last section raises a major problem of scale - how can you do justice to the Amiga's graphics capabilities without writing something as long as the average ROM kernel manual? As usual, we won't try to do this - instead the aim of this section is to take a "bottom up" approach, starting with an overview of Amiga graphics hardware, then showing how this is built on by the graphics, layers and Intuition libraries to create views, viewports, rastports, layers, screens, windows, and all the other hard work done by the Amiga to make things look easy to the user.

1. Amiga graphics hardware

The graphics hardware is generally one of the best understood aspects of the Amiga - this probably reflects the fact that the hardware manual is much more familiar territory to programmers coming to the Amiga from 8-bit machines like the 64 than the unfamiliar software concepts involved in the ROM kernal manuals. We shall therefore give only a brief overview here, without going into too much detail about fancy features (like "hold and modify") which will be found documented in many sources.

The graphics facilities of the Amiga are looked after by the custom chips Paula, Agnus and Denise - we agree that there is little point in worrying about the exact division of responsibility between these chips, and better to think of them as a single entity "the PAD". These chips share chip memory (the bottom 512K) with the 68000 on the Amiga, either in a civilised manner by accessing memory in alternate clock cycles while the 68000 is performing internal operations, or else in an uncivilised manner by blocking the 68000 from bus access until the PAD has finished. The amount of uncivilised access ("cycle stealing") depends on what the blitter is doing (see below), and on the screen resolution and number of colours. It is important to realise how much difference this can make - the amount of cycle stealing varies from zero for 16 colours lo-resolution display, to <u>all</u> of the display time for 16 colours high resolution, which means that the processor can only access the bus while the beam is off the screen. If you have "fast memory" (real fast memory - ie expansion memory) attached this isn't too much of a problem, as the 68000 is only blocked from the bottom 512K - and from the new "slow memory" at $C00000 - and can still access real fast memory at full speed. For an Amiga without expansion memory it can make quite a difference however. The relationship between the 68000, the PAD, and chip and fast memory is illustrated in figure 1.

1.1 Graphics hardware elements

The basic elements of the PAD's graphics capabilities are as follows:

The playfield hardware handles forming a basic display by accessing "bit-planes" (essentially screen RAM that can be located anywhere in the PAD's memory) and a "colour table" or "palette" indicating which of the Amiga's 4096 possible colours are to be used in the current display.

The sprite hardware handles moving up to eight independent graphic objects in front of the basic playfield. Hardware sprites are surprisingly limited on the Amiga compared with the 64; there are only eight of them, and they can be only up to 16 pixels wide, though they can be any height. These limitations are largely overcome by some cunning software making use of other Amiga hardware resources; sprites can be "joined together" to increase their effective width, and they can be "multiplexed" using the copper to increase the apparent number of them ("virtual sprites"). Sprites are mainly used for fairly simple, rapidly moving graphics objects on the Amiga - more complex (but slower moving) objects use "blitter objects" or "bobs".

The graphics coprocessor or copper is used to make changes to the display sychronised with the current beam position. This includes changing the bit-plane pointers telling the PAD where to fetch its graphics data, and changing the colour table values; it is this which is fundamentally responsible for the Amiga's ability to support overlapping "screens" (an Intuition concept) with different resolutions, colours etc, usually running different applications.

The blitter or bimmer is the Block Image Transfer (or, as Amiga now prefer, Block Image Manipulator) device which is responsible for most of the Amiga's animation capabilities. The blitter makes use of direct memory access (DMA) in Amiga chip memory; it takes input from three different "input streams", combines them in a choice of 256 different possible logical operations specified by "minterms", and outputs the result on an "output stream". Thus it is possible for example to take three input streams representing a graphics background, a mask, and a moving object, and output them into bit-planes corresponding to part of the current screen display - this is the basic technique used for animation with "bobs".

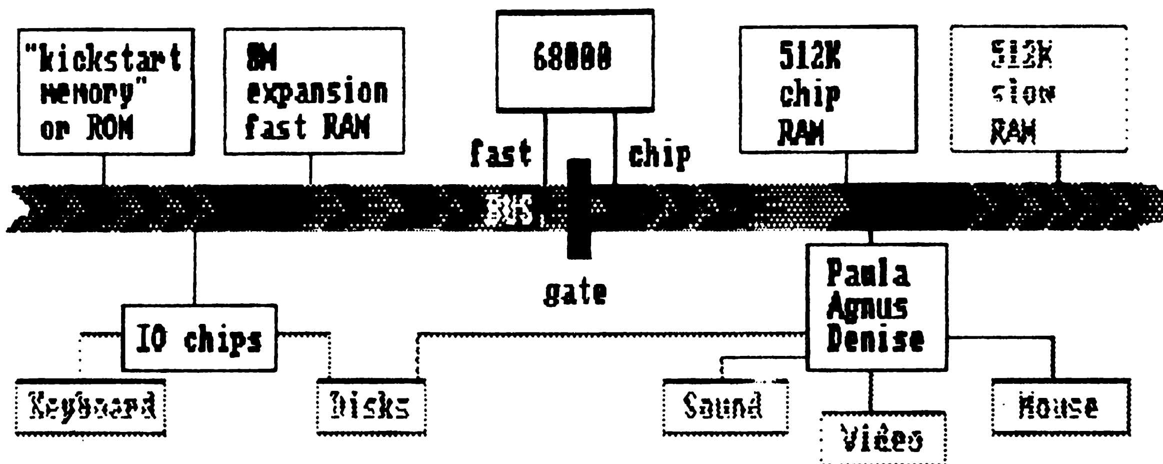


Figure 1 - Amiga Hardware Overview

Bitplanes - anwhere in chip memory -
select colour 0011

The PAD

BPL1PT
BPL2PT
BPL3PT
BPL4PT

col0
col1
col2
col3
col4
.
etc

0011

000010111

contents of
colour reg 3
determine
colour output
at this point

red - 0
green - 4
blue - 15

Figure 2 - PAD playfield access

Playfield access

The Amiga playfield hardware contains registers specifying the various graphics modes available (lo-res or hi-res, interlace on-or-off and "special effects" like dual playfield and hold-and-modify), pointers to up to 5 "bit-planes", and colour registers for up to 32 colours. The basic mechanism used by the PAD to determine what colour to display a given pixel is as illustrated in figure 2, and can be summarised as follows:

1. The PAD fetches one bit of information from each bit-plane and interprets the result as a binary number from 0 to 3 for 2 bit planes, 0 to 7 for 3 bit planes etc, up to 0 to 31 for 5 bit planes.

2. This number tells the PAD which colour register to use. Each colour register contains a 12-bit number specifying the amount of red (bits 8 to 11), green (bits 4 to 7) and blue (bits 0 to 3) in the final colour.

The fact that the bit planes specify a register number rather than an actual colour value is referred to as "colour indirection"; changing the colour register contents allows colours to be changed very rapidly, allowing a variety of effects, such as "colour cycle animation" as used in many Amiga paint packages and in the famous "bouncing ball" demo.

Note that the fact that the PAD registers contain pointers to the bit-planes in internal registers means that the bit-planes can be anywhere in chip memory, don't have to be next to each other, and can be in any order. This is reflected in an important graphics structure called a BitMap which contains pointers to the bit-planes used by any particular graphics object, as explained below.

The role of the Copper

The 64 contains a simple form of "graphics co-processing" in the form of "VIC chip interrupts"; the chip controlling the video keeps an eye on the position of the video beam, and is able to interrupt the processor when the beam reaches a specified position, allowing a number of "split screen" tricks to be implemented.

This is taken a very great deal further in the Amiga, in that the PAD contains a "graphics coprocessor" sychronised with the position of the video beam, running its own simple program which is known as a "copper list". This runs on the odd system cycles like the 68000 and unlike most of the PAD's DMA; as such it takes priority over the 68000, and over any odd-cycle access performed by a "nasty" blitter. The copper is capable of interrupting the CPU when the video beam reaches a specified position as in the 64; however, it is capable of much more than this, the most important new facility being the ability for the copper to modify

other registers in the PAD directly when the beam reaches a given position, allowing things like split-screen tricks to be implemented without any CPU intervention. Note that the copper can directly access only PAD registers; to affect memory in the rest of the system, it can either cause a 68000 interrupt and let the CPU do it, or it can do it more directly by writing to the registers which control the blitter.

(Allowing the copper to program the blitter and hence to affect external memory is potentially powerful but also very dangerous; for this reason the PAD copper control register contains a "danger bit", and the copper is prevented from accessing blitter control registers unless this bit is set. Using the copper and the blitter together also causes sychronisation difficulties, as the copper must wait for a "blitter finished" interrupt before doing trying to do anything else with the blitter - this is achieved using a "blitter finished disable bit" in the actual copper instructions.)

The copper has a distinctly limited instuction set, being limited to only three instructions, each of which occupies two words (one longword):

WAIT	till the beam reaches a specified position
MOVE	16-bits of data from the second word into a PAD register specified by the first word
SKIP	the next instruction if the video beam has passed a specified position

In a normal Intuition-managed Amiga display of several overlapping screens, the copper list will just consist of a series of WAIT and MOVE instructions, to set up the playfield hardware for the first screen to be displayed, wait until the beam reaches a position where the screens change, then adjust the playfield hardware for the next screen, etc.

The PAD contains a number of registers associated with the copper, an important one of which is a pointer telling the copper where to start processing its copper-list. The copper's "program counter" is automatically loaded with this value at the start of each vertical blanking interval - ie while the beam is off the screen ready to start a new frame. The copper then processes its instruction list as the beam travels down the screen; note that the list should be in a sensible order, so that an instruction to wait for vertical position 150 comes <u>after</u> an instruction to wait for vertical position 100, not <u>before it!</u> The copper list is usually terminated by an instruction to wait for the impossible such as line 255, column 254; the copper is rescued from this by the next vertical blank, which causes it to be automatically restarted at the start of its instruction list.

```
0001b4a8   2801  fffe   WAIT (0028,0000)  ;background till line 40
0001b4ac   0180  0777   MOVE (0180,0777)  ;col0    top vp = emacs
0001b4b0   0182  0fff   MOVE (0182,0fff)  ;col1    (1 bitplane med-res)
0001b4b4   01a0  0000   MOVE (01a0,0000)  ;col16   colours 16 to 19
0001b4b8   01a2  0d22   MOVE (01a2,0d22)  ;col17   are sprite 0
0001b4bc   01a4  0000   MOVE (01a4,0000)  ;col18    - intuition pointer
0001b4c0   01a6  0fca   MOVE (01a6,0fca)  ;col19
0001b4c4   01a8  0444   MOVE (01a8,0444)  ;col20   colours 20 to 31 are
0001b4c8   01aa  0555   MOVE (01aa,0555)  ;col21   set up for other sprites
0001b4cc   01ac  0666   MOVE (01ac,0666)  ;col22
0001b4d0   01ae  0777   MOVE (01ae,0777)  ;col23
0001b4d4   01b0  0888   MOVE (01b0,0888)  ;col24
0001b4d8   01b2  0999   MOVE (01b2,0999)  ;col25
0001b4dc   01b4  0aaa   MOVE (01b4,0aaa)  ;col26
0001b4e0   01b6  0bbb   MOVE (01b6,0bbb)  ;col27
0001b4e4   01b8  0ccc   MOVE (01b8,0ccc)  ;col28
0001b4e8   01ba  0ddd   MOVE (01ba,0ddd)  ;col29
0001b4ec   01bc  0eee   MOVE (01bc,0eee)  ;col30
0001b4f0   01be  0fff   MOVE (01be,0fff)  ;col31
0001b4f4   008e  0581   MOVE (008e,0581)  ;DIWSTRT - display window start
0001b4f8   0100  0302   MOVE (0100,0302)  ;BPLCON0 - bit plane display off
0001b4fc   0104  0024   MOVE (0104,0024)  ;BPLCON2 - bit plane priority
0001b500   0090  40c1   MOVE (0090,40c1)  ;DIWSTOP - display window stop
0001b504   0092  003c   MOVE (0092,003c)  ;DDFSTRT - display data fetch start
0001b508   0094  00d0   MOVE (0094,00d0)  ;DDFSTOP - display data fetch stop
0001b50c   0102  0000   MOVE (0102,0000)  ;BPLCON1 - bit plane scroll value
0001b510   0108  0000   MOVE (0108,0000)  ;BPL1MOD - odd bit plane modulo
0001b514   00e0  0001   MOVE (00e0,0001)  ;BPL1PTH - bitplane 1 at 1f780
0001b518   00e2  f780   MOVE (00e2,f780)  ;BPL1PTL
0001b51c   2901  fffe   WAIT (0029,0000)  ;wait for line 41
0001b520   0100  9302   MOVE (0100,9302)  ;BPLCON0 - bitplane display enabled
0001b524   9101  fffe   WAIT (0091,0000)  ;display emacs vp till line 145
0001b528   0100  0302   MOVE (0100,0302)  ;BPLCON0 - bitplane display off
0001b52c   9301  fffe   WAIT (0093,0000)  ;wait till line 147
0001b530   0180  0777   MOVE (0180,0777)  ;col0    next vp = workbench
0001b534   0182  0fff   MOVE (0182,0fff)  ;col1    (2 bitplane med-res)
0001b538   0184  0002   MOVE (0184,0002)  ;col2
0001b53c   0186  0f11   MOVE (0186,0f11)  ;col3
0001b540   01a0  0000   MOVE (01a0,0000)  ;col16   intuition pointer
0001b544   01a2  0d22   MOVE (01a2,0d22)
0001b548   01a4  0000   MOVE (01a4,0000)
0001b54c   01a6  0fca   MOVE (01a6,0fca)
0001b550   01a8  0444   MOVE (01a8,0444)  ;col20   other sprites
0001b554   01aa  0555   MOVE (01aa,0555)
0001b558   01ac  0666   MOVE (01ac,0666)
0001b55c   01ae  0777   MOVE (01ae,0777)
0001b560   01b0  0888   MOVE (01b0,0888)
0001b564   01b2  0999   MOVE (01b2,0999)
0001b568   01b4  0aaa   MOVE (01b4,0aaa)
0001b56c   01b6  0bbb   MOVE (01b6,0bbb)
0001b570   01b8  0ccc   MOVE (01b8,0ccc)
0001b574   01ba  0ddd   MOVE (01ba,0ddd)
0001b578   01bc  0eee   MOVE (01bc,0eee)
0001b57c   01be  0fff   MOVE (01be,0fff)  ;col31
0001b580   008e  0581   MOVE (008e,0581)  ;DIWSTRT
0001b584   0100  0302   MOVE (0100,0302)  ;BPLCON0
0001b588   0104  0024   MOVE (0104,0024)  ;BPLCON2
0001b58c   0090  40c1   MOVE (0090,40c1)  ;DIWSTOP
```

Listing 1 - Copper list disassembly

```
0001b590   0092  003c   MOVE (0092,003c)  ;DDFSTRT
0001b594   0094  00d0   MOVE (0094,00d0)  ;DDFSTOP
0001b598   0102  0000   MOVE (0102,0000)  ;BPLCON1
0001b59c   0108  0000   MOVE (0108,0000)  ;BPL1MOD
0001b5a0   010a  0000   MOVE (010a,0000)  ;BPL2MOD
0001b5a4   00e0  0001   MOVE (00e0,0001)  ;BPL1PTH bit plane 1 at 111b8
0001b5a8   00e2  11b8   MOVE (00e2,11b8)  ;BPL1PTL
0001b5ac   00e4  0001   MOVE (00e4,0001)  ;BPL2PTH bit plane 2 at 161b8
0001b5b0   00e6  61b8   MOVE (00e6,61b8)  ;BPL2PTL
0001b5b4   9401  fffe   WAIT (0094,0000)  ;wait till line 148
0001b5b8   0100  a302   MOVE (0100,a302)  ;BPLCON0 - bit plane display enab
0001b5bc   ffdf  fffe   WAIT (00ff,00de)  ;wait till line 255,col222
0001b5c0   3601  fffe   WAIT (0036,0000)  ;pardon?
0001b5c4   0100  0302   MOVE (0100,0302)  ;BPLCON0 - same as before???
0001b5c8   ffff  fffe   WAIT (00ff,00fe)  ;wait line line 255,col254
```

Copper list for two ViewPort display -
WorkBench screen pulled down in front of Emacs.

Listing 1 (continued)

Note that the use of the copper is not an optional extra in the Amiga, but is an integral part of the mechanism for even the simplest display. This is because the copper must be used to set up the bitplane pointers used by the playfield hardware at the start of every screen display; if this isn't done then the bit-plane pointers won't be reset, so the screen will go crazy after at most one single frame!

Listing 1 shows a copper list disassembly for a simple display of the workbench (2 bit-plane, med-resolution) "pulled down" in front of the MicroEMACs editor (1 bit-plane, med-resolution). The copper list disassembler program used to generate this is very simple, and is printed at the end of this section. You can use this to perform further experiments if you want; alternatively, a more powerful copper list disassembler can be found on the Fish disks.

1.4 Tricks with playfield hardware

The Amiga PAD contains a register containing an offset known as the modulo to be added to the PAD's bitplane pointers between each scan line and the next. This allows the actual bitplanes to be larger than the physical screen display (see Fig 3), which then forms a "window" (not in the Intuition sense) onto the larger display. It can be seen that adjusting the bitplane base address and the modulo allows a variety of smooth-scroll tricks to be implemented.

The modulo is also involved in interlace mode on the Amiga. In this mode, the apparent vertical resolution is doubled by first outputting a screen consisting of every second line in the display, then outputting another screen consisting of the "missing" lines, offset by half a screen scan-line. Missing every second line is achieved by setting a modulo of 40 (40 bytes corresponds to 40*8 = 320 pixels for a low-res screen with interlace); showing first one screen then the other is achieved by having two copper lists, which are used for alternate frames.

Two special modes available from the Amiga hardware are "dual playfield", and "hold and modify" modes, each involving up to six bit-planes. In dual playfield, the bitplanes are divided into two groups, one of which appears "behind" the other - ie the "back" playfield is displayed where the "front" playfield is showing background (colour register 0). In hold-and-modify (HAM) mode, two bitplanes specify either normal colour selection using the remaining bitplanes, or else a colour value (red green or blue) to "hold and modify" - the remaining bitplanes then specify the new value of this register. Further information about these modes can be found in the hardware and other manuals.

1.5 More about the blitter

While the blitter is principally used for graphics image manipulation, it is in fact generally useful for operations involving blocks of memory in the bottom 512K - it is much faster at this sort of thing than the 68000 - and is therefore also used for non-graphics purposes, such as rapid motion of data in disk buffers. The blitter can operate either in "nice" mode in which it uses alternate clock cycles not needed by the 68000, or in "nasty" mode in which it forces the 68000 off the bus until it has finished - this is not an unreasonable thing to do during block transfers, at which the blitter is much more efficient than the 68000. Note that the existence of "nasty mode" is another good reason for the division of "chip" and "fast" memory.

Besides its essential function of moving blocks of memory around (and perhaps modifying them in the process), the blitter has another mode in which it can be used to generate lines and area fills directly into bitplane memory. This secondary function of the blitter is used by the graphics library line-drawing and area-fill routines.

2. Graphics system software

The graphics capabilities of the Amiga are controlled by three libraries of routines as follows:

graphics.library - controls the blitter and copper in a way consistent with the Amiga's multi-tasking environment. Builds on the basic graphics capabilities to provide a variety of routines for drawing and animating the display.

layers.library - provides some of the routines needed to treat a display as a number of overlapping "layers", as used for example by Intuition Windows. The distinction between the graphics and layers libraries is fairly arbitrary, and was probably a matter of development convenience as much as anything else. This is reflected in the fact that most of the drawing routines in the graphics libraries pay attention to layers structures if present, and, conversely, the layers routines make use of lower-lever concepts called "regions", which are themselves manipulated using routines from the graphics library.

intuition.library - provides a consistent interface to the user by building on routines in the graphics and layers libraries to provide things like Screens and Windows. Appears both as a library which can be invoked by application software, and as an "input handler" in the handler-chain maintained by the input device (mouse keyboard etc input coordinator); in its latter incarnation is capable of handling things like window moving, resizing etc in a way transparent to the application program, and of sending messages to the application program telling it things of interest to it using Intuition Direct Communication Message Ports (IDCMPs).

Note that in the process of providing a standard interface, the Intuition library introduces some restrictions - eg to standard screen sizes.

The graphics library

The Amiga graphics library can be viewed as consisting of various levels as follows:

- level 0 - elementary playfield control. Rasters and BitMaps.
- level 1 - elementary copper control. Views and ViewPorts
- level 2 - elementary sprite control. Simple sprites.
- level 3 - elementary blitter control.
- level 4 - non-layered drawing primitives. Non-layered RastPorts.
- level 5 - layered drawing primitives. Regions, layers, and layered RastPorts.
- level 6 - elementary animation. GELs (graphics elements) consisting of virtual sprites (VSprites) and blitter objects (Bobs)
- level 7 - complex animation. Animobs.

In the rest of this section we will try and give an overview of levels 0 to 5; discussions of animation will be found in many sources, including ROM kernel manual volume 2.

Elementary playfield control

Probably the lowest level graphics structures available are the ColorMap containing a pointer to a ColorTable, which itself contains the actual colour register values used by part of an Amiga playfield, and the BitMap containing pointers to the actual bit planes used in the display; these bitplanes are also known as the "raster". The BitMap structure is as follows:

BitMap:
- Number of bytes per row
- Number of rows
- Flags
- Number of bit planes (depth)
- Padding byte
- Up to eight pointers to bit planes.

Note that this structure contains some redundancy; the largest number of bitplanes used by an ordinary Amiga display is 5, and the largest used by special modes such as "dual playfield" is 6; however the BitMap structure supports up to 8. The idea of this is to support upward compatibility with future-Amigas-that-may-be-to-come with even more graphics capability - there is a lot of this kind of redundancy in the graphics and Intuition structures.

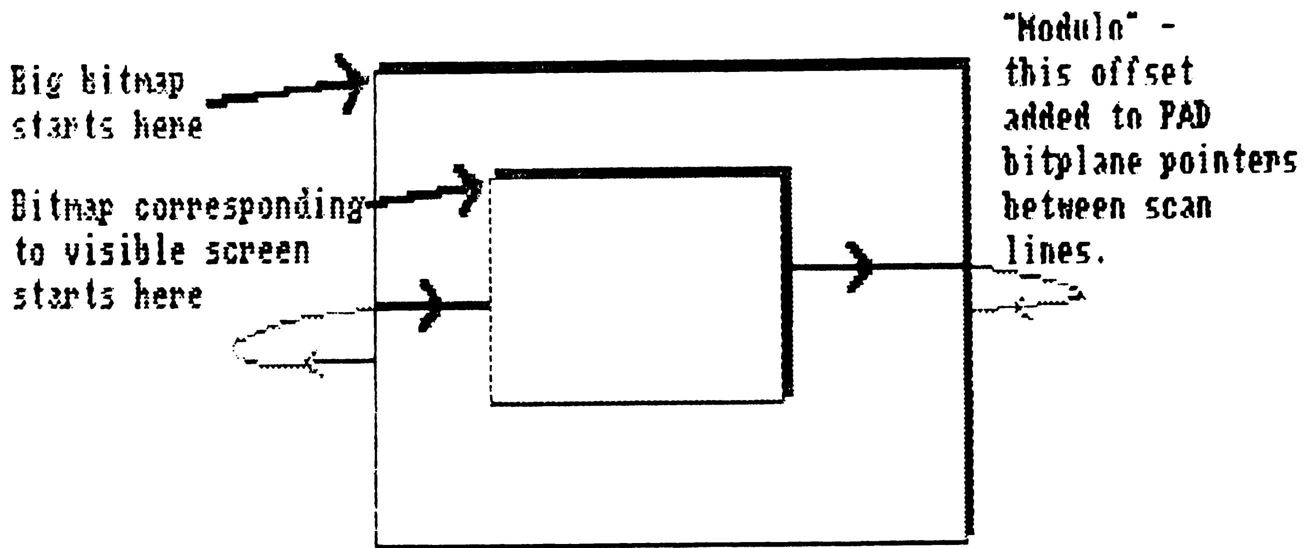


Figure 3 - Big bitplanes and Modulo

View
ViewPort software
ViewPort
ColourMap
ColourTable
RasInfo
BitMap
Bitplanes
The PAD.
Copper changes PAD pointers between V-Ports
Linked list of ViewPorts
ViewPort
ColourMap
ColourTable
RasInfo
BitMap
Bitplanes

Figure 4 - Views and viewports

It is for the same reason that the ColorMap structure points to the actual hardware ColorTable rather than simply containing register values; future Amigas with different colour hardware should be supportable provided user software goes through the routines provided (SetRGB4CM etc), and doesn't attempt to access the ColorTable directly.

Note the following further points about BitMaps:

1. BitMap structures are used to represent the actual "screen RAM" currently in use by the Amiga, eg the current Intuition Screen. However, they can also be used to indicate sub-elements (such as Windows) within the current display, and for bits of display which aren't currently on the screen, such as the obscured part of a "smart refresh" Window. This can be confusing!

2. As mentioned above, the use of the modulo facility in the PAD allows the actual bitplanes to be larger than the physical screen display (see Fig 3), which then forms a window onto the larger display. The offsets to be used for this kind of trick are held in another structure called RasInfo, which itself points to the BitMap.

Copper control from the graphics library - Views and ViewPorts

As mentioned above, the Amiga copper can be used to change the PAD playfield registers part of the way through the screen display, allowing the screen to be divided into a number of separate horizontal slices (corresponding to bits of different Screens under Intuition). Each horizontal slice is associated with a graphics structure called a ViewPort as follows:

```
ViewPort: pointer to next ViewPort in display
          pointer to ColorMap structure for this slice
          pointer to copper list to set up display for this slice
          pointer to copper list to set up sprites for this slice
          pointer to copper list to get rid of sprites after this slice
          pointer to user copper list
          width and height of this slice
          x and y offset of this slice
          modes - interlace, dual playfield on/off etc
          reserved
          pointer to RasInfo for this viewport
```

The various ViewPorts are tied together in a linked list, headed by a structure called a View:

```
View:     pointer to first ViewPort in this display
          pointer to hardware copper list for this display
          (pointer to second copper list for alternate frames in
                    interlaced display)
          x,y offsets for whole display
          modes - overall interlace on/off, genlock on/off, etc
```

The relationship between View, ViewPorts, ColorMaps, RasInfo and BitMaps is illustrated in figure 4. The ViewPort and View structures are very much concerned with copper list manipulation; each ViewPort contains pointers to up to four "intermediate copper lists" containing copper instructions relating to the playfield, to sprites (two lists), and to any special copper tricks wanted by the user. These intermediate lists of instructions are then merged together (with WAIT instructions in a sensible order), and linked with the lists corresponding to the other ViewPorts in the current View to produce the final "real" copper list for actual execution by the copper hardware.

4.1 Graphics library routines using ViewPorts

The graphics library routines concerned with ViewPorts can be divided into categories as follows:

1. Copperlist manipulation. The lowest level graphics library routines involved in copper list and ViewPort management are the routines and macros concerned with setting up and adding instructions to copper lists directly; these are useful when constructing "user copper lists".

2. Creating ViewPorts. A number of routines are provided to allocate and deallocate memory used by the ViewPorts (BitMaps, ColorMaps etc), to initialise the View and ViewPort structures themselves, and to actual create and use the copper list corresponding to a given view.

3. Manipulating ViewPorts. Once a display has been created, it can be manipulated by smooth-scrolling ViewPorts (by manipulating RasInfo), and/or by changing colour values used in ViewPorts.

4. Miscellaneous routines exist to read current beam position (though by the time you get this it may be out of date due to task switching!), to wait for top of next video frame, and to wait till the beam reaches the bottom of a specified ViewPort.

5. Getting rid of ViewPorts. When a ViewPort is no longer needed - eg because a new View has been loaded - memory used in creating its various sub-structures can be deallocated using various routines to free memory for its copper lists and bitplanes.

See Table 1 for a summary of graphics routines concerned with ViewPorts.

Table 1 - Graphics library routines for ViewPorts

Copperlist manipulation

CINIT (macro)	- initialise intermediate copper list
CBump	- increment copper list pointer
CWAIT (macro)	- insert WAIT instruction then CBump
CMOVE (macro)	- insert MOVE instruction then CBump
CEND (macro)	- insert WAIT for position 255,254 to terminate list

Creating ViewPorts

InitBitMap	- initialise a BitMap structure
GetColourMap	- allocate memory for ColorMap and ColourTable
SetRGB4CM	- set one colour in the ColourTable structure pointed at by this ColourMap.
InitView	- initialise a View structure to default values
InitViewPort	- initialise a ViewPort structure to default values
MakeVPort	- construct an intermediate copper list to set up the display for this ViewPort
MrgCop	- merge together all the intermediate copper lists for the various ViewPorts into one "real" copper list for the whole View
LoadView	- actually make the copper execute the copper list for this View

Manipulating ViewPorts

ScrollVPort	- change current copper list to reflect new ViewPort RasInfo - used for "smooth scroll"
SetRGB4	- set a colour reg value for specified ViewPort.
LoadRGB4	- set all colour reg values from a table.
GetRGB4	- read a colour reg value for specified ViewPort.

Miscellaneous

VBeamPos	- return vertical beam position
WaitTOF	- wait for top of next video frame
WaitBOVP	- wait till beam reaches bottom of ViewPort.

Getting rid of ViewPorts

FreeCopList	- free memory for an intermediate copper list
FreeVPortCopLists	- free memory for all intermediate copper lists associated with ViewPort
FreeCprList	- free memory for "real" hardware copper list
FreeColourMap	- free memory for ColourMap and ColourTable
FreeRaster	- free memory for a bitplane

4.2 ViewPorts and Screens

As mentioned above, the graphics concept of a ViewPort is very closely linked to the Intuition concept of a Screen; Intuition Screen structures actually contain ViewPort structures. However, it is useful to realise the distinction. An Intuition Screen is a fairly high level concept and has a title, push and pop gadgets etc; it may be considered to be "behind" other screens, even to the extent of not being currently visible on the display at all. A ViewPort on the other hand corresponds strictly to a horizontal slice of the current display, which may (but doesn't have to be) part of an Intuition screen. The relationship between Intuition's notion of things and the same setup as seen by the graphics library is illustrated in Figure 5, for a setup involving three screens, one of which has been pulled down, and one of which is currently entirely obscured.

5. Elementary sprite control

The mechanisms used for elementary sprite control on the Amiga are known as "simple sprites"; this is in contrast with the more complex mechanism used in the animation system (GELs) which are known as "virtual sprites".

Both simple and virtual sprites are controlled using the copper, using two additional intermediate copper lists associated with each ViewPort; however, the mechanism used to control virtual sprites is much more complex, and is known as the "virtual sprite machine". Simple sprites are "stolen" from this machine - ie individual hardware sprites are flagged as no longer available for use with the virtual sprite machine, but are allocated instead to an application program. The principal structure used in this connection is the SimpleSprite

```
SimpleSprite:  pointer to data used for sprite control
               height of simple sprite
               current x,y position of simple sprite
               number of simple sprite (0 to 7)
```

Routines used to control simple sprites are as follows:

```
GetSprite     - attempt to allocate hardware sprite for your
                exclusive use
ChangeSprite - change appearance of sprite
MoveSprite    - change position of sprite relative to given
                ViewPort, or the current View.
FreeSprite    - return sprite for use by others
```

6. Elementary blitter control

At the lowest level, the graphics library provides elementary contention management, to prevent one task trying to do something with the blitter, while another task is in the middle of doing something else. The simplest routines which handle this are the following:

- OwnBlitter - go to sleep until the blitter is free for your exclusive use. Should be followed by a WaitBlit.
- WaitBlit - wait until the blitter is really free - ie until it has finished all internal operations involved in a blit currently in progress.
- DisownBlitter - release blitter for use by others.

Less anti-social are routines which allow you to queue a request for a blit, using a structure called a "bltnode":

bltnode:
- pointer to next bltnode, set up by QBlit or QSBlit
- pointer to your blitter-hitting function.
- flag indicates if cleanup function needed
- synch position for synchronised blit
- pointer to cleanup function (optional)

These routines are as follows:

- QBlit - queue an ordinary (not beam synchronised) blitter request
- QSBlit - queue a synchronised blitter request - ie one to be done when the video beam reaches a specified position. Takes priority over non-synchronised blits.

When your bltnode is processed, your blitter-hitting function is called repeatedly until it returns a zero; the cleanup function is then called if specified. This allows multiple blits - eg blits involving several bit-planes - to be performed as closely as possible together.

6.1 Bottom line Amiga graphics

Once you have got hold of a ViewPort and some BitPlanes to write into (the easiest way of doing this is to open a Screen using the new SCREENQUIET flag in Intuition), and once you know how to control sprites and the blitter using the elementary routines discussed above, then you are in a position to start writing Amiga graphics applications. This is appropriate in situations where you want maximum possible speed, but aren't worried about fancy features like the "layered RastPorts" discussed below - eg for games applications. It is important to realise that this provides a mechanism for hitting the "screen RAM" and the blitter hardware directly for maximum performance, but in a way which is legal and won't screw up the rest of the system.

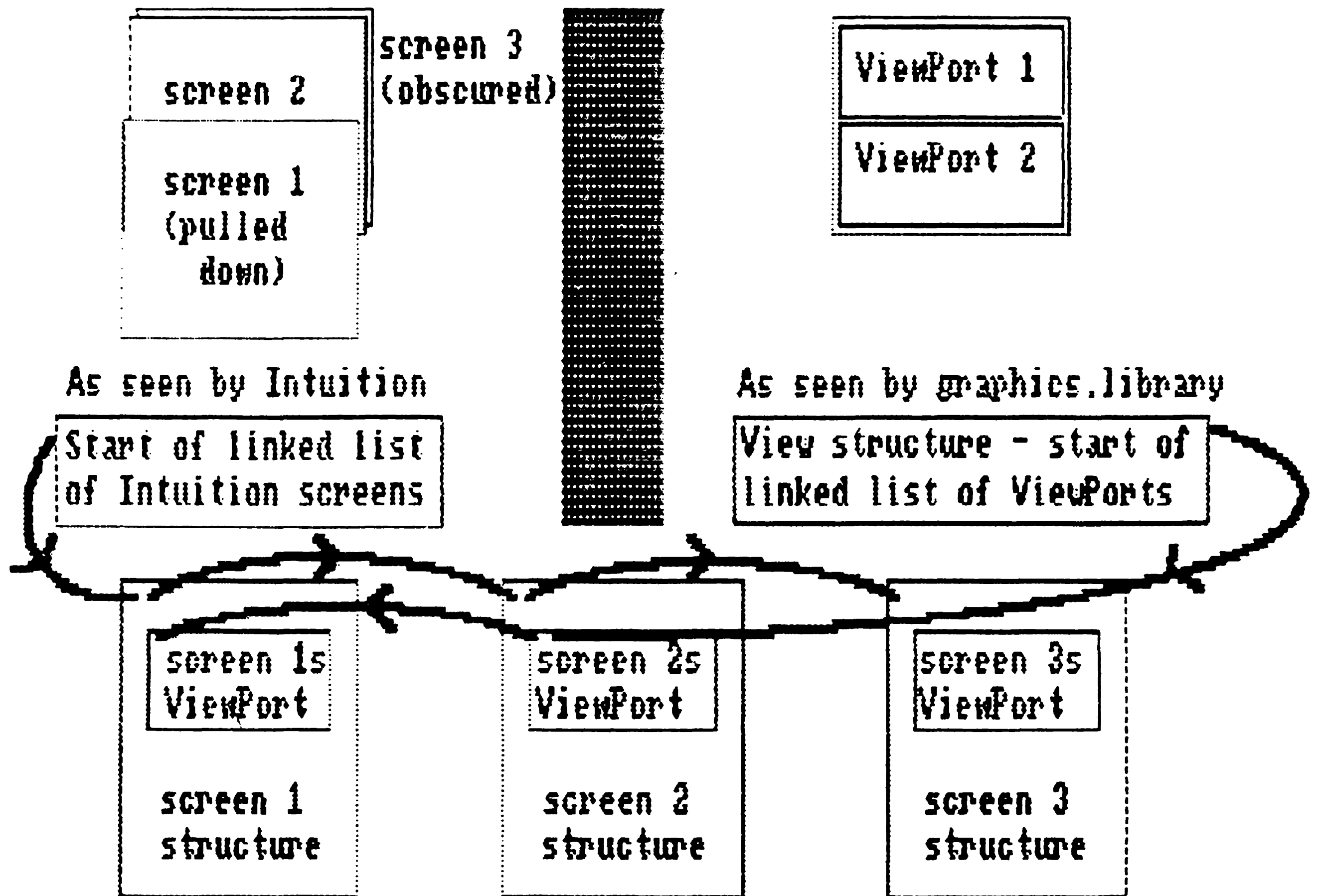


Figure 5 - ViewPorts and Screens

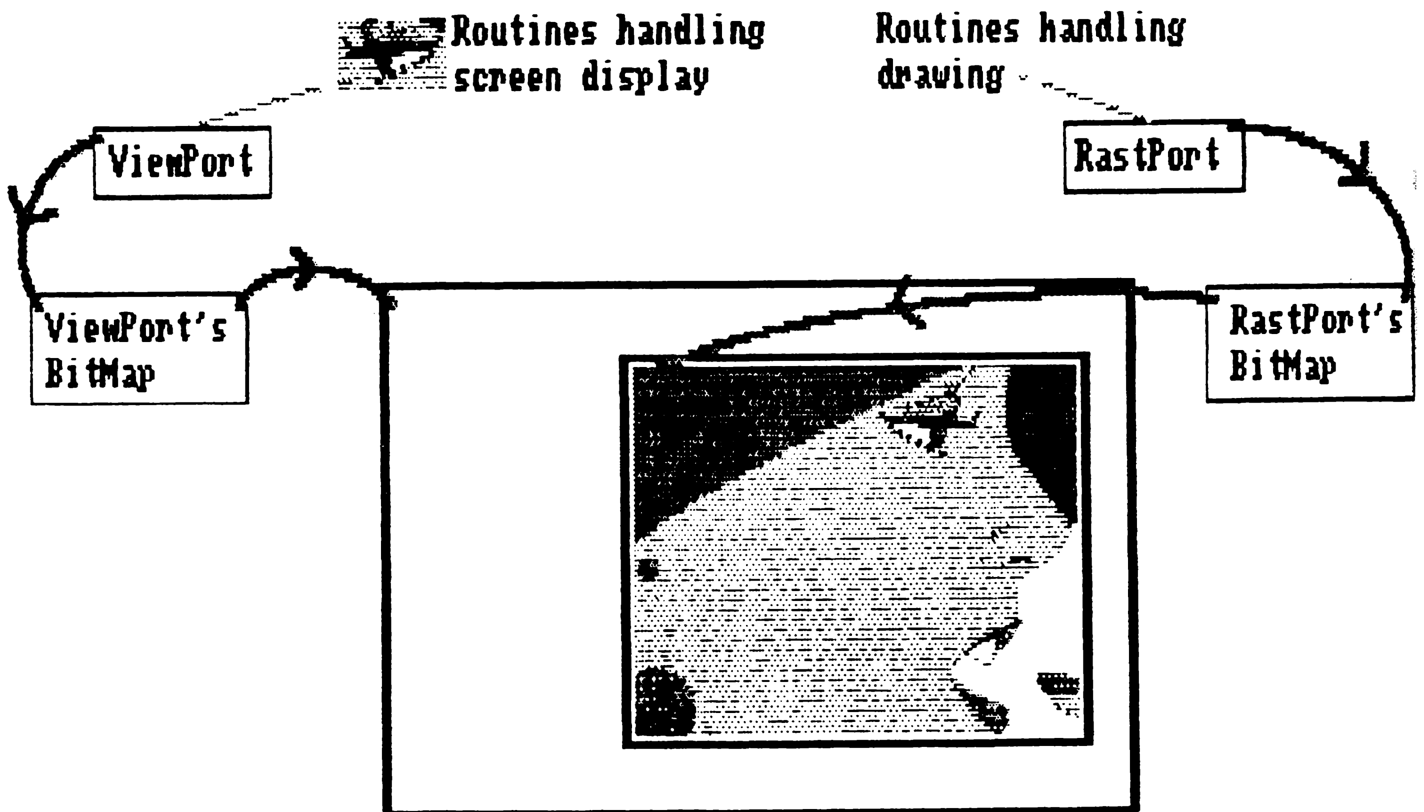


Figure 6 - ViewPorts and RastPorts

Non-layered drawing primitives - non-layered RastPorts

The basic system structure concerned with copper-list manipulation and the display of graphics is the ViewPort; the basic structure consisting with the actual creation of graphics images (drawing or "rendering") is the RastPort. If you are doing all your own drawing by directly accessing ViewPort bitplanes using the CPU or the blitter as discussed above, then you don't need to worry about these; however, if you want to use any of the system's drawing routines, then you need a RastPort.

The first item in a RastPort is a pointer to a Layer structure; this is crucial in determining the behaviour of the RastPort. Generally speaking, if this pointer is NULL, then the graphics routines assume that they don't need to worry about layers (eg overlapping windows), and that they don't need to worry about clipping - ie if an effort is made to draw outside the RastPort's associated BitMap (see below), then the graphics routines will go ahead and draw, causing unknown damage! However, if you don't mind handling your own clipping, and if you don't require the full capability of drawing into overlapping windows, then this is much faster than drawing to a layered RastPort as discussed below, and provides a good compromise between handling everything yourself, and going through the full "layered RastPort" approach, which handles all sorts of things for you, but has some time overhead.

Of equal importance is another pointer to a BitMap structure, telling the graphics routines where to draw to. This structure is the same as that used by ViewPorts; a RastPort's BitMap may be exactly the same as a ViewPort's; it may be a subset of it if you want to draw into just part of a ViewPort, or it may not correspond to part of a current ViewPort at all - eg if you want to draw to a region of chip memory which is currently off the visible screen, then swap it in later. A possible relation between a ViewPort's and a RastPort's BitMaps is illustrated in figure 6. There is an obvious relation between RastPorts and Windows; however be warned in advance a Window is in fact a layered RastPort which means that life is a bit more complicated than shown in figure 6, as explained below!

"Pens" and RastPorts

The notion of a pen is used in two contexts in a RastPort. In the simpler case, "pen number" is used simply to specify a colour register number to use in a drawing operation - ie what number (bit pattern) to to "JAM" in the bitplanes. However, the RastPort contains three special "pens" for drawing operations as follows:

FgPen (APen) - foreground pen using for drawing operations. The RastPort remembers a current colour (pen number) for FgPen, and a current pen position.
BgPen (BPen) - background pen (colour register number) used when JAMing both foreground and background into bitplanes (drawing mode JAM2)
AOLPen (OPen) - pen (colour register number) used for area-fill operations.

7.2 RastPort structure

Stuff of interest in a RastPort structure is as follows:

RastPort:

- Pointer to Layer structure for layered RastPort
- Pointer to BitMap structure - where to draw to
- Pointer to pattern to use for area fills
- Pointer to "TmpRas" - temp storage for area fill, text, etc.
- Pointer to "AreaInfo" - info for area fill operations
- Pointer to "GelsInfo" for animation
- Write mask
- Foreground pen - which colour reg to use for foregrounds
- Background pen - which colour reg to use for backgrounds
- Areafill pen - which colour reg to use for block (area) operatio
- DrawMode - put in foreground only (JAM1), foreground and background (JAM2), xor drawing (COMPLEMENT) or invert (INVERSEVID)
- Number of words in areafill pattern
- Flags
- Line pattern for textured lines
- Current "pen" position for move, draw etc
- Current "minterms" for blitting - reflect current DrawMode etc
- Current pen width and height
- Pointer to current text font
- Text style, flags, height, width, baseline position & spacing

7.3 Graphics library routines using RastPorts

All the Amiga graphics drawing ("rendering") routines make use of RastPorts - there are quite a lot of these (see Table 2). The routines can be sub-divided as follows:

1. Initialising RastPorts. Two routines are available to initialise a RastPort to default settings, and to initialise a "TmpRas" structure, containing a pointer to an area of memory for use as a buffer area for area fill, flood fill, or text.

2. Drawing into RastPorts. Basic drawing routine for points, lines, circles and ellipses. Work equally well into non-layered or layered RastPorts.

3. Area fill operations. "Area fill" routines normally use the blitter to flood an area with a given pattern (pointed to by AreaPtrn pointer in the RastPort), the area being defined as a series of vectors or end-points specifying the (polygon) shape to be filled; there are also routines for rapid filling of ellipses and circles, though this involves more work by the 68000. Area fills use a structure called AreaInfo:

   ```
   AreaInfo: pointer to start of table of vectors
             pointer to current vertex (end-point) in table
             pointer to start of flags table
             pointer to flags for area fill
             number of end points in list
             maximum number of end points allowed
             x,y for first point in this polygon
   ```

   Besides the "standard" area fill routines using this structure, there are special fast routines for area filling rectangles, and a slow routine for "flood filling" an arbitrary area on the screen up to a boundary - this involves a lot of work by the 68000 searching bitplanes for boundaries, and is much slower than "area filling" a known shape using the blitter.

   Note that in order to use area fill, two additional structures are needed associated with the RastPort - an AreaInfo structure as above, plus a "TmpRas" (temporary raster - also used for text) big enough to hold the area being filled. There is no problem adding these structures to Intuition's Windows' RastPorts (see below). Routines InitTmpRas() and InitArea() can be used to initialise these structures - note that the latter is called with the address of the AreaInfo structure, NOT - as claimed in some versions of the documentation - the address of the RastPort. See the 1.2 autodocs for full information.

4. Blitting into RastPorts. Besides the contention-management routines (OwnBlit etc) mentioned above, the lowest-level access to the block-copy/block-modify facilities of the blitter simply allow you to blit directly from one BitPlane to another, with no clipping and entirely at your own risk - this is called BltBitMap. Higher level routines handle clipping in a layered RastPort (ClipBlit etc), and a variety of special effects such as blitting through a stencil.

5. Text into RastPorts Text is just a special sort of graphics on the Amiga. Routines which render text into RastPorts make use of a structure called a TextFont which describes a particular font to the system in terms of the actual shapes of the characters, and a structure called TextAttr which describes it in terms of name, size and style; text fonts are loaded into chip memory where they are linked into a system list of currently resident fonts, and where they remain until got rid of. Further font-handling routines relate to the management of the 'fonts:' directory on disk; these are in a special library 'diskfont.library'.

   The graphics library text routines are fairly elementary and are built on by other software in the Amiga system, notably by Intuition which provides a convenient mechanism for positioning text in windows ("IntuiText"), and by the console device, which uses the basic text-handling routines with the system's default font to provide an emulation of a full ANSI standard terminal, with cursor control, standard "escape" sequences, etc.

See table 2 for a summary of graphics routines concerned with RastPorts.

Table 2 - graphics routines concerned with RastPorts

Initialising RastPorts

```
InitRastPort    - set Mask, FgPen, AOLPen and LinePtrn to -1,
                  DrawMode to JAM2 and font to standard.
InitTmpRas      - set up a TmpRas structure pointing to a buffer
                  for temp storage for area fill, flood and text.
```

Drawing into RastPorts

```
WritePixel      - set pixel specified to PenA colour
ReadPixel       - return colour reg number at pixel specified
Move            - move "pen" position within RastPort
SetAPen         - select FgPen colour
SetBPen         - select BgPen colour
SetOPen (macro)- select AOLPen colour
SetDrMode       - select drawing mode JAM1, JAM2, COMPLEMENT,
                  or INVERSVID
Draw            - draw line from current to new pen position
PolyDraw        - draw a series of lines into RastPort
DrawEllipse     - draw ellipse or circle into RastPort
```

Area fill in RastPorts

```
InitArea        - initialise table of vectors used to store end-
                  points for area fill
AreaMove        - start a new list of end points for area fill
AreaDraw        - add another to list of end points for area fill
AreaEllipse     - put an ellipse in the buffer used for area fill
AreaCircle (macro)  - put a circle in the buffer used for area fill
AreaEnd         - process the list of end points, and do the fill
Rectfill        - fast area fill rectangular area
SetRast         - fast area fill entire RastPort
Flood           - slow flood area around point up to boundary
```

Blitting into RastPorts

```
BltBitMap       - blit from one bitmap to another with no clipping.
BltClear        - zero a region of memory using the blitter.
BltBitMapRastPort - blit from a bitmap into a RastPort
BltMaskBitMapRastPort - blit from a bitmap through a mask into a
                  RastPort.
BltPattern      - blit a pattern (like area fill) into a RastPort,
                  though an (optional) mask.
BltTemplate     - blit a template mask into a RastPort.
ClipBlit        - blit from one RastPort into another, paying
                  attention to layers (if present).
ScrollRaster    - blit from one RastPort into itself, to perform a
                  scrolling operation.
```

Table 2 (continued)

Text into RastPorts

AddFont	- adds a text font structure to the system font list
RemFont	- removes a font from the system font list
OpenFont	- search the system font list for a given TextAttr and indicate font in use
CloseFont	- indicate font no longer in use
SetFont	- sets the font pointer and text attributes of a given RastPort
AskFont	- gets text attributes of given RastPort
SetSoftStyle	- set soft style (algorithmically generated effects like bold or italic) for given RastPort
AskSoftStyle	- get soft style for given RastPort
Text	- write specified text at current pen position
TextLength	- figure out how long text is going to be - eg is it going to fit in the RastPort?
ClearEOL	- clear to right edge of RastPort, according to current text height and baseline
ClearScreen	- perform ClearEOL, then clear from there to the bottom of the RastPort

Routines in diskfont.library

AvailFonts	- build a list of information about all fonts currently available in RAM or on disk, including their TextAttr structures
OpenDiskFont	- like OpenFont, but loads from disk if font not in current system font list

8. Layered drawing - Layered RastPorts

As mentioned above, most of the Amiga drawing routines will work either into a non-layered RastPort with a NULL Layer pointer, or into a layered RastPort associated with a Layer structure, controlled by a number of routines in the layers library.

Layers in the Amiga serve two closely related purposes:

1. They allow a single BitMap - such as the one associated with a particular Intuition Screen - to be considered as a number of independent overlapping drawing areas (RastPorts) - such as Intuition Windows.

2. They allow a single BitMap to be shared by a number of different tasks, and provide the locking and unlocking routines necessary to prevent the different tasks interfering with each other. Typically, this is achieved transparently by each task having its own Window(s), and hence its own layered RastPort(s); the Layers routines then apply the necessary locks internally when performing functions where windows could interfere with each other.

8.1 Layers and Windows

In the same way that graphics ViewPorts are closely related to Intuition Screens, so layered RastPorts are very closely related to Intuition Windows. An Intuition Window contains a pointer to its "very own" layered RastPort and hence to an associated Layer structure; the simplest way of getting hold of a layered RastPort is therefore to open an Intuition window, then use

```
rp = window->RPort;
```

It is then possible to use most of the graphics routines discussed above to draw into the window, and Intuition's facilities to depth-arrange windows, with no overhead over using the Layers facilities more directly.

In fact, it is possible for an Intuition Window to have either one or two layers associated with it. A normal window (with its origin at the top left of the title bar) has a single layered RastPort and Layer associated with it; this means that when drawing to the windows RastPort, you have to apply your own "mini-clipping" to avoid drawing over the window borders and system gadgets. A GIMMEZEROZERO window on the other hand has two layers associated with it; the borders and system gadgets have their own "private" RastPort and Layer so that you can't draw over them, while the RastPort and Layer you get from window->RPort correspond to the "inside" of the window, with its origin at the top left of the inside (non-border part) of the window. (This is convenient, but has a fairly high overhead; the 1.2 graphics library documentation suggests getting the same effect by using InstallClipRegion to mask the borders out of

routines which draw into the window, and ScrollLayer to specify an offset for drawing routines to compensate for the presence of window borders.)

8.2 Layers structures

Layers logic on the Amiga has been updated since the original release 1.0, and again between releases 1.1 and 1.2, most notably by improving the mechanism used for locking and unlocking Layers, which now uses a system of flags called Semaphores. Some old structures, and routines like InitLayers ThinLayerInfo and FattenLayerInfo, remain for reasons of downward compatibility with old versions - however, we shall not bother to consider these in this section.

Layers logic on the Amiga is looked after using the following principal structures. Layers are held together in a linked list; a Layer_Info structure contains a pointer to the current top Layer in a BitMap, which in turn contains a pointer to the next Layer, and so so. These structures are as follows:

1. A Layer_Info structure is associated with the BitMap being shared - eg with an Intuition Screen which is to be split into a number of Windows - and contains a pointer to the current top Layer within the BitMap. It also contains other general information about the layers arrangement within the BitMap such as the number of layers currently locked; and Semaphore information allowing the Layers_Info structure itself to be locked and unlocked, to avoid contention problems when creating new Layers within the BitMap. A new Layer_Info structure can have memory allocated and be initialised using a layers.library routine NewLayerInfo.

2. A Layer structure is associated with a layered RastPort - eg with one particular Window within a Screen's overall BitMap. A layered RastPort containing a pointer to a Layer structure, and the Layer structure itself, are created together using routines from the layers library CreateUpfrontLayer and CreateBehindLayer. Once created, layered RastPorts are used in the drawing routines just like non-layered RastPorts; however the drawing routines will note the non-zero Layers pointer, and will perform clipping (and clever tricks like drawing into currently obscured areas) as necessary. Useful elements in this structure are as follows:

```
Layer:    Pointers to Layer in front, Layer behind
          Pointer to list of ClipRects for this Layer
          Pointer back to this Layer's RastPort
          MinX, MinY, MaxX, MaxY bounds for this Layer
          Flags - Layer type (see below),
                  Layer-needs-refreshing flag
          Pointer to SuperBitMap, if SuperBitMap Layer
          Pointer to list of ClipRects for SuperBitMap for
               SuperBitMap Layer, OR pointer to damage list
               ClipRects for simple refresh Layer
          Various stuff used by system for Layer locking
          Various stuff used by system for Layer refresh
          Pointer to damage list regions for simple refresh
```

3. A ClipRect structure is associated with one particular rectangular area of a Layer for purposes of clipping - eg with a corner of a Layer which is currently obscured by another Layer (eg another Window) in front of it. Each Layer can be broken into a series of rectangles on or off the visible screen for purposes of clipping; this is reflected in the fact that the Layer structure contains a pointer to a linked list of ClipRect structures as follows:

```
ClipRect: Pointer to next ClipRect
          Pointer to previous ClipRect
          Pointer back to Layer owning this ClipRect
          Pointer to BitMap for this ClipRect
          MinX, MinY, MaxX, MaxY bounds for this ClipRect
          Various stuff for system use
```

   Note that each ClipRect has its own associated BitMap structure; this may correspond to part of the currently visible screen, or (if we are dealing with a "smart refresh" or "super-bitmap" Layer) to bitplanes which are buffer areas corresponding to currently obscured regions of the Layer. It is by drawing into such areas that the graphics library drawing functions are able to draw into currently "hidden" areas of Layers and Windows.

4. Regions and RegionRectangles perform very similar functions to Layers and ClipRects, but contain the information in a purely geometric format, which saves time over using the full ClipRect structures. The main function of Regions is to create a DamageList showing which areas of a Layer may need re-drawing; this can then be converted to an equivalent list of ClipRects when the time comes to actually do the re-drawing. The damage list contains a Region structure followed by a series of RegionRectangles:

```
Region:   MinX, MinY, MaxX, MaxY for this region
          Pointer to first RegionRectangle

RegionRectangle:    Pointers to next, previous RegionRectangles
                    MinX, MinY, MaxX, MaxY for this RegionRectangle
```

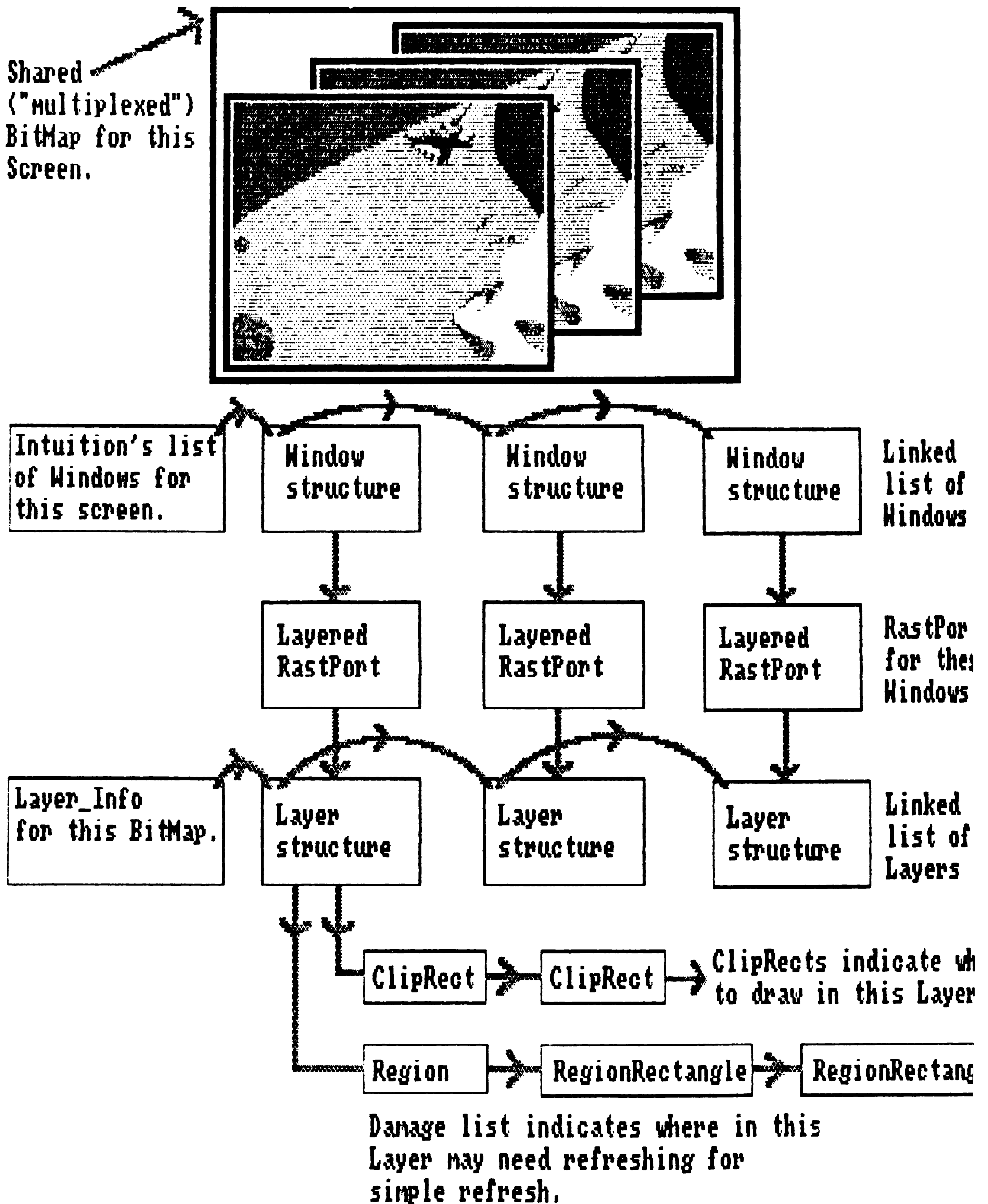


Figure 7 - Layered RastPorts

The relationship between these various structures is illustrated in figure 7.

8.3 Types of Layers

There are three principal types of layer, distinguished by flags in the Layer structure, corresponding to different mechanisms for "refreshing" the layer - ie of correcting any "damage" done to the layer due to other things going on in the BitMap, such as other layers being moved in front of it. These three Layer types correspond to the three principal Intuition Window types, and are as follows.

In many ways, the most ingenious form of layer is the "simple refresh" or LAYERSIMPLE variety. Simple refresh layers are divided into ClipRects corresponding to only the visible parts of the layer; no buffer areas (BitMaps) are provided for parts of the Layers not immediately visible. Instead it is up to the application to correct the damage, assisted by the system as follows:

1. If something happens (such as movement of another layer) that may cause part of this Layer to be have to be refreshed, then

   a. The DamageList (Region and RegionRectangles) is updated to indicate which parts of the Layer may need redrawing, and

   b. A flag LAYERREFRESH is set in the Layer structure, to show that it needs refreshing.

2. When the application notices that LAYERREFRESH is set, it proceeds as follows:

   a. Calls a function BeginUpdate, which saves the Layer's current ClipRects, then works out a new set of ClipRects from the DamageList.

   b. Redraws the whole layer; however, because we are using the damage-list ClipRects, this means that only the parts of the layer that were damaged will be redrawn, so the system won't waste time re-drawing stuff which is all right anyway.

   c. Calls a function EndUpdate, which restores the Layer's normal ClipRects.

   Note that if you are working through Intuition, then you can arrange to get an IDCMP message REFRESHWINDOW when your window's layer needs refreshing; you can then perform refresh using Intuition functions BeginRefresh and EndRefresh, which are functionally equivalent to BeginUpdate and EndUpdate but using Window structures rather than

working directly with Layers.

The next type of layer is the "smart refresh" or LAYERSMART variety. In this sort of layer, the system will allocate memory for an off-screen buffer area when part of the layer becomes obscured, and this off-screen area will be included in the Layer's list of ClipRects, so that the graphics routines will draw into it if necessary. When the off-screen area becomes visible again, the system deals with copying this area back into the visible screen bitplanes and deallocates the buffer memory; the application therefore doesn't have to worry, unless the layer is resized.

The final type of layer is the "super bitmap", obtained by setting both LAYERSMART and LAYERSUPER flags in the Layer structure. In this case, the layer has an off-screen buffer area permanently allocated to it (you have to provide this when creating the layer), which can be bigger than the current layer size, or indeed bigger than the entire screen display if you want this. In the case of a SuperBitMap layer, the Layer's ClipRects point either into the visible screen BitMap, or into the SuperBitMap, depending on whether the corresponding part of the Layer is currently visible or not; the graphics routines therefore update either the screen BitMap or the SuperBitmap accordingly. The layers routines then deal with updating the screen BitMap from the SuperBitMap or vice-versa, as parts of the layer become revealed or hidden due to layer re-arrangement or resizing; this continues until the layer is deleted, when the system performs a final update of the SuperBitMap, leaving it reflecting the last known state of the layer.

Note that keeping the visible part of the display "synchronised" with the SuperBitMap will be handled automatically if you use the system drawing routines; however, this won't be the case if you choose to make changes in either the Screen bitplanes or the SuperBitMap bitplanes directly. If you decide to do this, then two routines exist to update the SuperBitMap from the screen and vice versa; these are called SynchSBitMap and CopySBitMap respectively.

Like all the Layers facilities, SuperBitMaps are best accessed by way of Intuition Windows. Note that a Window making use of SuperBitMaps should also be a GIMMEZEROZERO Window - with its borders gadgets etc in a separate layer - so that only the main data area of the window is associated with the SuperBitMap.

Two additional bits in the Layers flags which are used independently of the Layer type discussed above are as follows:

LAYERBACKDROP - indicates that the Layer has to stay at the back, and prevents sizing or depth-arranging. Used for Intuition "back-drop windows".

LAYERREFRESH - flags simple Layer needs refreshing.

Table 3 - Routines Concerned with Layers and Regions

Layers library - initialisation

NewLayerInfo	- alloc and init memory for Layer_Info structure
DisposeLayerInfo	- free memory associated with Layer_Info struct
CreateUpfrontLayer	- create new Layer & layered RastPort at front
CreateBehindLayer	- create new Layer & layered RastPort at back
DeleteLayer	- delete Layer & free memory

Layers library - locking

LockLayerInfo	- wait till Layer_Info free, then lock it
UnLockLayerInfo	- unlock Layer_Info
LockLayer	- wait till Layer free, then lock it
UnlockLayer	- unlock Layer
LockLayers	- lock Layer_Info then all layers
Unlocklayers	- unlock all layers then Layer_Info

Layers library - arrangement

UpfrontLayer	- move specified Layer to front
MoveLayerInFrontOf	- move specified Layer in front of another specified Layer
BehindLayer	- move specified Layer to back
MoveLayer	- move Layer to a new position in BitMap
SizeLayer	- change size of this Layer

Layers library - update

BeginUpdate	- convert DamageList to ClipRect list, and swap into Layer's ClipRect pointer
EndUpdate	- restore Layer's ClipRect pointer to normal

Layers library - miscellaneous

InstallClipRegion	- install specified Region in layer for clipping
ScrollLayer	- scroll around a SuperBitMap, or specify a plotting offset in a non-SuperBitMap layer
SwapBitsRastPortClipRect	- swap bits between RastPort and ClipRect (used for pull-down menus)
WhichLayer	- return pointer to Layer which specified point is in

Table 3 (continued)

Graphics library - Regions

NewRegion	- alloc & init memory for Region structure
DisposeRegion	- free memory for all RegionRectangles & Region
OrRectRegion	- add anything new from rectangle to Region
OrRegionRegion	- add anything new from region1 to region2
AndRectRegion	- clip away Region outside specified rectangle
AndRegionRegion	- clip away Region2 outside Region1
ClearRectRegion	- clip away Region inside specified rectangle
ClearRegion	- clip away everything from Region
XorRectRegion	- add bits of rectangle to Region if not there already, remove if there already
XorRegionRegion	- add bits of Region1 to Region2 is not there already, remove if there already.

Graphics library - Locking

AttemptLockLayerRom	- attempt to lock Layer; fail if already locked
LockLayerRom	- wait till Layer free, then lock it
UnlockLayerRom	- unlock Layer

Graphics library - miscellaneous

CopySBitMap	- update Layer window from SuperBitMap
SynchSBitMap	- update SuperBitMap from Layer window

.4 Routines using Layers

As mentioned above, there is little real distinction between the graphics and layers libraries; it is therefore convenient to consider the two together. Routines involving layers can be subdivided as follows:

1. Layers initialisation. Routines exist to create and get rid of Layer_Info structures, and to add or get rid of Layer structures from the linked list of Layers attached to the Layer_Info, either at the front or the back of any Layers already present.

2. Layer locking. Before adding new layers, or performing other operations involving the Layers list such as depth-arranging, it is necessary to "lock" the Layer_Info structure, to avoid inter-task contention. This is usually looked after by the other Layers routines as necessary; you only need to lock Layer_Info yourself if you are going to access the Layers structures directly, or if you are planning to lock more than one Layer - if so, you should first lock Layer_Info, to prevent a possible inter-task "deadlock". Similarly, when performing operations on a Layer's ClipRects, it is necessary to "lock" the Layer itself; again this is usually handled by the system as necessary. Finally, in some cases such as certain Intuition operations such as pull-down menus, it is necessary to lock all layers from graphics output for a while.

3. Layer arrangement. Routines exist to move layers to the front, to move them to the back, to move them in front of another specified layer, to move them "sideways" in the BitMap, and to change layer size.

4. Layer update. Routines BeginUpdate and EndUpdate used for "simple refresh".

5. Layer miscellaneous. Misc routines are InstallClipRect allowing you to install a specified Region in a layer for special clipping, ScrollLayer which EITHER moves a SuperBitMap around in a layer, OR specifies an offset for plotting in non-SuperBitMap layers, WhichLayer which reports which layer a given point is in, and SwapBitsRastPortClipRect (!), which provides a quick way of getting up menus in an otherwise locked layered display.

6. Graphics regions. Routines used in damage-list management, to update a Region (list of RegionRectangles) by performing logical operations between Regions and rectangles (eg "anything inside this rectangle and not already in the Region should be added to the Region"), or between Regions and other Regions. Can be used for to create drawing masks for "special effects"; a user-supplied Region can be "installed" in a Layer using InstallClipRegion, or by pointing to as a "pseudo DamageList" and calling

BeginUpdate.

7. Graphics locking Two routines LockLayerRom and UnlockLayerRom precisely equivalent to the layers locking routines in the Layers library, plus a routine AttemptLockLayerRom.

8. Graphics miscellaneous. Two routines CopySBitMap and SynchSBitMap for use when doing your "own" SuperBitMap management, eg when writing directly into SuperBitMap bitplanes.

See table 3 for a summary of routines concerned with Layers and Regions.

References

The main references used for this section have been the Amiga Hardware Manual, and ROM Kernel Manual Volume 2 - though the latter is now rather out of date in some respects, and also contains some errors. For up-to-date information, see the 1.2 h-files, and the 1.2 routine descriptions.

```
/**********************************************

   copper list disassembler for kickstart

   9th March 87 Ariadne Software Ltd

**********************************************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <graphics/view.h>

struct copins {          /* copper list instruction */
   UWORD ir1;
   UWORD ir2;
};

struct IntuitionBase *IntuitionBase;
struct copins *copptr;

main()
{
   UWORD first,second;

   if ((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library",0)) == NULL) exit(FALSE);

   printf("Copper list disassembly\n\n");

   copptr = (struct copins *)(IntuitionBase->ViewLord.LOFCprList->start);

   do {
      first = copptr->ir1;
      second = copptr->ir2;

      printf("%08lx   %04x  %04x   ",copptr,first,second);

      if (!(first & 0x0001)) {

         printf("MOVE (%04x,%04x)\n",(first & 0x01FE),second);

      } else {

         if (!(second & 0x0001)) printf("WAIT"); else printf("SKIP");
         printf(" (%04x,%04x)\n",
                      (first & 0xFF00)>>8,(first & 0x00FE));
      }

      copptr++;
      } while (!((first == 0xFFFF) && (second == 0xFFFE)));

   CloseLibrary(IntuitionBase);
   return(TRUE);
}
```

Appendix - Simple copper list disassembler

Part III - Getting started in C

Section I - Introducing C

Many Amiga owners who want to go beyond programming in Amiga BASIC will be interested in the language "C". This is the language that was used to develop a lot of the Amiga system software such as the graphics library, and the "Intuition" software which provides pull-down menus, windows, and the rest of the Amiga user interface. C has also been used to develop many important Amiga application packages such as paint packages, animation packages, word processors, music packages, data bases, and even fast arcade-type games.

However, C has a reputation for being a "difficult" language, not to mention expensive! In this part of the Kickstart Guide we look at what C is, and how it differs from programming in more familiar languages like BASIC. We shall start by discussing the strengths and weaknesses of C, then look briefly at some of the peculiarities of the language. We will then look at various aspects of C in more detail.

People who are high-level programmers by nature tend not to like C very much. Compared even with BASIC it is quite weak, particularly as regards string-handling. It is also inclined to be cryptic, and tends to make very heavy use of "pointers" - these are variables which hold the addresses where other variables are stored in memory (more about this later), which is something which assembler programmers are used to worrying about, but isn't something you expect in a high level language. By making extensive use of pointers, it is possible to write really phenomenally incomprehensible code in C!

C is also distinctly lackadaisical about "typing". In C, like in Pascal, you have to declare all variables before you use them, and have to say what "type" of thing they are - integer, long integer, floating point, character, string or whatever. In Pascal, having declared a variable to be of a certain type, it has to stay that way; C however is "weakly typed", in that it is possible to convert types at the drop of a hat using an operation known as "casting". This can also easily lead to confusion, and is something else that high-level programmers tend to object to - we would recommend that they investigate something like Modula 2 as a probably far more congenial alternative.

So, what about low level programmers, approaching the Amiga from a background programming 8-bit machines like the Commodore 64 in assembler? People from this background usually start by finding C alien and unforgiving; their first few efforts give umpteen error messages and apparently spurious warnings ("pointers do not point to the same object"), and they end up being appalled by the size of "object module" they get from an apparently trivial program - it is possible to end up with over 12K of code for a program to print your name on the screen, if you're not careful! Assembler programmers will probably find it easier to learn the

68000 and program the Amiga using an assembler, which they will find to be powerful, but overall quite familiar.

Advantages of C

So is there any point bothering to learn C, when there seems to be so much going against it? In a lot of cases, the answer to this question is "Yes - very definitely!" If you want to write arcade games, or any application in which your main interest is wizzing bits about in bit planes as quickly as possible (bit planes are as close as the Amiga comes to having "screen RAM"), then you should probably learn 68000 assembler. If you want to write something like an accountancy package, you should consider learning Modula 2. However, if you want both speed of execution, and quite a lot of the advantages of using a high level language, then C provides a very good compromise - it is a sort of "half-way house" or "middle level" language. So if you need speed, but are also doing "serious" programming, in which you are liable to end up with rather a lot of code interconnected in a tricky manner, then you will find C has the following advantages.

1. Viewed as a "super-assembler", C has a lot going for it. If you have some appreciation of what your C compiler is actually doing, then it is possible to write code in C with suprisingly little overhead, either in code-size or in execution speed. It is also quite easy to drop into assembler from C, for those routines where extra speed is really vital.

2. C lends itself much better than assembler to writing in a "structured" manner, with independent intercommunicating functions. It is quite easy to build your own "library" of useful functions, and to make use of libraries from other sources such as "standard libraries" available from Lattice and Amiga, which can cut development time dramatically. Contrary to some popular belief, it is possible to write C in a quite readable way, which also aids maintainance and debugging.

3. C is very much the native language on the Amiga, since it is what so much of the system software was written in! Using C, it is particularly easy to make full use of the facilities provided in the "ROM kernel"; these are generally of a high standard, and it's silly not to take advantage of them.

4. C is supposed to be a portable language, though in practice this is a lot more trouble than it should be. However, if portability is important to you, C will at least ease the task of moving your programs across a variety of machines, especially if you are careful about using "standard" functions.

5. The main advantage of C is the time-factor. Assembler programmers on large projects who have switched to C report that after a few months experience, they can be up to ten times more productive generating working debugged code. True, the resulting program may run a bit slower than if it was written in assembler

- but is it really worth the development time penalty?

Getting to C from BASIC or assembler

If you are approaching C on the Amiga from a background programming 8-bit machines in BASIC or assembler, you will find that things are very different.

The first thing you have to get used to is that C is compiled then linked - instead of just typing in a program and typing "run", or even running a two-pass assembler, there are a lot more stages to go through. We will be looking at these stages in detail later - however an overview is of the whole process is more or less as follows:

1. First of all you have to type in your "source-code" using some form of editor; there is a simple screen-editor on the standard Amiga Workbench which can be entered by typing "ED" in at the CLI. (The CLI is the Command Language Interface - an icon for this can be made to appear in the system drawer by switching CLI ON in Preferences. We shall assume some familiarity with the CLI - if you don't know about it then a number of books are available which explain it, including the official AmigaDOS manual.) Other editors are available from a variety of sources - a favourite used by many Amiga programmers is MicroEMACS. A C program consists of a series of "functions" - more about these later - which in turn call other functions in the same module, in some other module, or in a "system library". The system library functions perform standard operations such as printing a string, so you can't do very much without them!

2. Next, you need to save your source code, and possibly to exit the editor - though if you are using an editor like EMACS and have enough room in memory you may prefer to leave it running for use again later. You now need to run your C compiler - if you are using the Lattice compiler this runs in two phases, going through an intermediate stage known as a "q-file" (for "quad") which you will probably keep in RAM-disk, and ending up with "object code" (o-file).

3. The compiler may give you various error messages, in which case it will refuse to produce an o-file, and you need to go back to step 1 and re-edit. It may also give you "warning" messages - eg if it thinks you are trying to use a variable without having initialised it sensibly. It is up to you to decide if you are going to pay attention to warning messages (and go back to step 1) or whether you can safely ignore them.

4. The compiler o-file is not yet in a runnable form, because it will contain references to functions and variables defined in other modules, or in standard libraries. In fact, the o-file has a complicated format, and contains a variety of record-types. Some of these contain a sort of "partial" 68000 code without much in the way of address information; others contain information

about functions and variables defined in other modules; others still contain "relocation" information about how to adjust the code depending on where in memory it ends up.

To resolve references to other modules etc, another program has to be run called a "linker". Typically, the linker is used to join together some sort of standard "start-up" module, a number of modules comprising your program, and routines for standard functions selected from one or more "libraries". If the linker succeeds - ie if it manages to find all the functions and variables it needs - then it produces an AmigaDOS "load file"; otherwise it will give error messages, and you need to go back to step 1.

5. It is worth noting that on the Amiga, the load file still doesn't contain absolute 68000 code. Instead it contains a number of records known as "hunks" each of which corresponds to one bit of code or data, and which contain fields known as "blocks" containing code, relocation information enabling the code to be loaded anywhere in memory there happens to be room for it, and (optionally) "symbolic" information allowing text labels etc from your original program to be passed to a "symbolic debugger" such as "Wack" or "Metascope".

   The actual process of "scatter-loading" the load file is made to happen by typing the program name at the CLI; it can also be arranged to happen which you click on an icon on the Workbench. AmigaDOS will decide where in memory to put the various hunks, then will load, relocating as it goes; it will then execute from the start of the first hunk, which should be the startup module. You can now determin whether your code works or not - if not go back to step 1!

Peculiarities of C

C was developed as a systems programming language, designed to make the process of writing things like operating systems quicker and more efficient. It is descended from BCPL, which has never caught on the same extent, though which still has its enthusiasts. Both BCPL and C have been used to write operating systems; BCPL was used to write Tripos, and C was developed along with Unix. All these systems are very relevant to the Amiga; AmigaDOS is written in BCPL and is based on Tripos, while most of the ROM kernel (apart from the lowest levels of Exec) is written in C, and was developed on Unix systems.

Unix was originally developed on the PDP11; C was therefore developed with a number of features designed for the generation of efficient compiled code using the PDP11 instruction set. This instruction set has been very influential on microprocessor design, including the 68000, which accounts for the continuing popularity of C, due to its (comparitively) efficient code generation. The fact that C is designed in this way also accounts for its unsatisfactory character when viewed as a high

level language; it also accounts for a number of distinctive peculiarities of the language. These include the existence of pre- and post- increment and decrement operators, which appear in statements like

```
nextchar = string[index++];
```

which picks up the next character in a string then increments the index; this has a direct equivalent in the 68000 instruction set in the form of register indirection with postincrement, eg

```
MOVE.B    (An)+,Dn
```

which moves the byte at the address in register An to data register Dn, and increments An to point at the next byte. Similar peculiarities are the existence of operators like "+=" ("value += 4;" is equivalent to "value = value + 4;"), and the fact that assignment statements have values, the value being the result of the assignment, ie the thing in front of the equals sign - since this value will be hanging around in a register, the compiler might as well use it! This latter peculiarity is frequently used to assign something a value and test it in the same line, as in the highly characteristic

```
if ( (buffer=AllocMem(1000,0)) == 0) printf("No room\n");
```

Note the difference between "=" meaning assignment - being used here to set a variable "buffer" equal to the value returned by function AllocMem() - and "==" meaning the equality operator, being used to check if buffer ends up as zero. Forgetting this is a classic C "beginner's error".

Perhaps the most important aspect of C viewed as a "super assembler" is the language's facilities for structures and pointers. A C-pointer is almost (but not quite) the same as a machine address; the statement

```
struct Node *ln_Succ;
```

declares that "ln_Succ" is a "pointer" to a structure called a Node, ie it contains the address where the Node is in memory. The pointer is not quite the same as a machine address, because the statement

```
ln_Succ++;
```

will not increment ln_Succ by one byte (or one word or one long-word); it will increment it by an amount equal to the size of the structure called "Node". For this reason (amongst others), C-compilers are very fussy about being told what sort of objects pointers point to, and give you irritating warning messages if you forget to.

A "structure" itself is a collection of the various data-types supported by C, including other structures and pointers; thus a "Node" structure is defined

```
struct Node {
     struct Node *ln_Succ;      /* pointer to next node */
     struct Node *ln_Pred;      /* pointer to previous node */
     UBYTE ln_Type;             /* type - unsigned byte */
     BYTE ln_Pri;               /* priority - signed byte */
     char *ln_Name;             /* pointer to node name */
};
```

A "node" is in fact a standard structure used by all the Amiga system software - its definition can be found in an "h-file" exec/nodes.h, which should be provided with your C compiler. You can include this in your compilation using an instruction called #include; assuming you have done this, you can then declare a node-structure for your own use by

```
     struct Node mynode;
```

or declare a pointer to a Node by

```
     struct Node *mypointer;
```

You can then pick up some part of the structure - say the node priority - directly by

```
     mypriority = mynode.ln_Pri;
```

or indirectly by

```
     priority = mypointer->ln_Pri;
```

Again, there is a very close correspondence between this C facility and a 68000 address mode - in this case register indirection with displacement,

```
     MOVE.B     d16(An),Dn
```

where address register An is being used as a pointer to the start of a structure, and offset d16 is being applied to this value to pick up a byte this far past the start of the structure, and move it into address register Dn. The use of structures and pointers is crucial to C programming, which is why we have mentioned them in this introduction; we shall look at how they work in detail later.

A simple example

A classic simple C program is as follows:

```
/* a classic example */

main()
{
     printf("Hello Amiga!\n");
}
```

This uses a standard library function printf() to output the string "Hello Amiga!" followed by a new-line, represented by the escape sequence "\n". This program consists of the definition of a single function main() which takes no parameters; by convention C execution is transferred to a function called main() as soon as startup operations have been completed. Further points to note about this example are the use of /* and */ pairs to enclose comments, the use of braces { and } to enclose functions, and the use of semi-colon as a line terminator. Forgetting semi-colons is another classic "beginner's error"; carriage returns have no special significance in a C program, and in fact any amount of "white space" consisting of spaces, tabs and carriage returns is taken as equivalent to a single space. This gives considerable latitude in the layout of a C program; the system of indentation adopted in this book is however fairly standard, and is recommended as an aid to program legibility.

Accessing a library

In order to follow a slightly more complicated example which makes use of one of the built-in functions from the Amiga's Kickstart ROM, it is necessary to understand a little about "libraries" on the Amiga.

A "library" is essentially a group of routines accessed via a jump table at the start, a bit like the "kernal" on the Commodore 64. However, the Amiga differs from the 64 in having a large number of libraries concerned with different aspects of the machine - examples that we will be using here are the Exec library which handles interrupts, multi-tasking etc, the "Intuition" library which provides a standard user-interface, and the DOS library which gives access to the functions provided by AmigaDOS. A second crucial difference from machines like the 64 is that Amiga libraries don't necessarilly always live at the same place in memory; before using a library it is necessary to "open" it, and find out where it is.

A simple example using the Intuition library is as follows; this will flash the Amiga display rapidly six times.

```
/* Intuition screen beep */

#include <exec/types.h>

extern APTR OpenLibrary();
extern VOID CloseLibrary(),DisplayBeep();

APTR IntuitionBase;          /* pointer to intuition library */

main()
{
     int i;

     IntuitionBase = OpenLibrary("intuition.library",29);

     if (IntuitionBase != 0) {     /* if open succeeded... */

          for (i = 0;i < 6;i++) {  /* then perform six... */
               DisplayBeep(0);     /* screen flashes... */
               Delay(5);           /* and pauses... */
          }

          CloseLibrary(IntuitionBase);  /* then close */

     }                             /* end of if */
}                                  /* end of function main() */
```

This example is much more complex, and contains a number of aspects of C which we will be discussing in detail later. However, some explanation may be useful at this stage:

1. The **#include** at the start is not a line of C - note the absence of a terminating semi-colon! Instead it is an instruction to a part of the compiler known as the "macro pre-processor" to include a header file exec/types.h, which contains definitions of some standard data-types such as APTR and VOID. Things defined as "macros" like this are by convention distinguished by being given in capital letters.

2. The #include is followed by **extern** statements telling the compiler a bit about functions used in this module which are defined elsewhere - in this case in the library file amiga.lib. Here we are using a function OpenLibrary() which returns an address pointer (APTR) and two functions CloseLibrary() and DisplayBeep() which don't return anything (VOID).

3. Following this we have the declaration of a "global variable" which is an APTR called IntuitionBase, where we are going to store a pointer telling us where the Intuition library is in memory. The names of global variables like this are output to the object file, so that IntuitionBase will be accessible to the routine in amiga.lib which needs it.

4. Our function main() starts with another variable declaration - this is a "local variable" called "i" which will be used as a

counter only within this function. Variables declared within functions like this are known properly as "automatic variables" since they are created automatically when the function is called, and got rid of when the function exits.

5. The first thing we do is to attempt to open a library called "intuition.library", with version number greater or equal to 29. OpenLibrary is itself a routine from the Exec library; we don't have to worry about opening the Exec library, since the Exec and DOS libraries are opened for us by the standard startup modules.

6. If OpenLibrary succeeds it returns a pointer telling us where intuition.library is in memory; if it fails it returns zero. The rest of the program is therefore conditional on this call not returning zero; "!=" is "not equals" in C.

7. We now use a for loop to beep the screen 6 times. The "for" construction is very powerful in C, and contains specifications for loop initialistion (i = 0), "keep looping" condition (i < 6), and what to do between loops (i++ - ie increment i).

8. Within the loop we call the Intuition function DisplayBeep with a zero parameter, telling Intuition to flash the entire display rather than one specific "screen". The function DisplayBeep() is actually in amiga.lib and the necessary code will be incorporated at link time; what it does is to set up processor registers appropriately, then figure out what address it should be calling by applying an offset to the value held in the IntuitionBase variable; it then transfers execution to this address, and hence to the Kickstart ROM.

9. Following the call to DisplayBeep, we call a routine from the DOS library Delay() with a parameter of 5, telling it to delay for 5 fiftieths of a second.

10. When we have finished, we must "close" the Intuition library to inform the system that this task doesn't require Intuition any longer; to do this we call an Exec function CloseLibrary().

C on the Amiga

There are a number of versions of C available for the Amiga; the "standard" version which was originally circulated to developers by Commodore was Lattice C V3.03; updates of this, Lattice C V3.1 and V4.0 are now available from dealers. Another version which has found favour with many developers is Manx Aztec C. We will base our examples on Lattice - however, if you are using another compiler, you should find things are similar in principle.

The original Amiga linker was called Alink (for "Alan's linker" not "Amiga linker" -after a programmer for MetaComco). This came in versions 1.0 and 1.1, the latter boasting a keywork FASTER which had the effect of speeding matters up considerably. An alternative linker called Blink was developed by a group called

"The Software Distillery" in the States and placed initially on the Public Domain - this is faster than the original Alink without the FASTER keyword, and boasts more fancy facilities; Blink is now provided as standard by Lattice.

Full instructions for setting up a C development environment on Amiga should be included with the compiler; details will vary depending on what version of what compiler you are using, on whether you have hard disk or extension RAM, and on personal preference. If you are using Lattice, then the process will be something like the following:

You will probably need to create a "stripped" Workbench disk, with enough room on it for the compiler, linker and utilities. Start by chucking out the demos, clock, calculator, notepad etc, then ditch all fonts, all printer drivers except your printer, all keymaps except your country, all AmigaDOS commands you don't use, and any libraries and devices you don't need like translator.library and narrator.device. Then install at least both phases of the compiler lc1 and lc2, and the linker blink - the best place to put these is probably in the C: directory.

Adjust the startup file s:startup-sequence so that it assigns the following logical device names:

LC: where to find lc1, lc2 and blink. Probably assigned to C:

INCLUDE: where to look for system .h files given in angle-brackets (eg #include <intuition/intuition.h>) Probably assigned to a volume name like h_files: if you are working on floppy, to a directory name like dh0:Lattice/h_files if you are working on hard disk; can be assigned to RAM: if you have lots of memory and don't mind the time it takes to copy all the h-files into RAM-disk when you boot; alternatively, might be assigned to a recoverable RAM-disk.

LIB: where to look for linker scanned libraries like lc.lib and amiga.lib. Probably assigned to INCLUDE:lib

QUAD: where to put the intermediate quad file between lc1 and lc2. Almost certainly assigned to RAM:

You are now in a position to try a simple example, such as the Intuition screen-beep program given above. Put yourself in a suitable directory C-progs (or something), and makedir a sub-directory obj to hold the object files. You can now create a source-file beep.c using ED or EMACS, then compile and link it using something like the following:

```
stack 15000
LC:lc1  -iINCLUDE:  -oQUAD:  beep
LC:lc2  -oobj/beep.o  QUAD:beep
LC:blink  FROM LIB:c.o+obj/beep.o  TO beep
                                      LIBRARY LIB:lc.lib+LIB:amiga.
```

The first thing we do is to increase the stack-size given to a new program - this is important because Lattice needs more than the system default which is only 4K. The next two lines invoke the two phases of the compiler, telling it search INCLUDE: for any "#include <>" files, to put its intermediate file in QUAD:, and to put its final object file in directory obj. The third line - which should be entered as all one line in the CLI - tells the linker to link Lattice's standard startup module c.o with object module obj/beep.o to produce a load module beep, looking first in lc.lib then in amiga.lib for library functions such as OpenLibrary() and DisplayBeep().

Note that it is possible to shorten this considerably by making use of the Lattice program lc which will invoke both compiler phases and (optionally) also the linker, and/or by making use of AmigaDOS "execute" files. Note also that this simple example is untypical in that our whole program is in one object file - in a real application we would want to have things split up so that functions relating to menu-handling (say) were in one file, functions relating to gadgets in another, drawing functions in a third or whatever; we would then join these all together with a startup module such as c.o using the linker. Knowing how to split functions between different source-files is a bit of an acquired skill - beginners generally start by having everything in one huge source-file (which takes forever to compile), then go to the opposite extreme of having every function in a separate source-file; the ideal is probably somewhere in between these.

Section 2 - Elements of C

Introduction

We will now consider the principal types of object encountered in C programs, in the form of functions, variables, and fundamental data types. This is not inclusive - our intention is not write an "alternative reference guide" for Amiga C, and we make no claim to cover all aspects of the language - we hope however that it may serve as a useful introduction.

When discussing aspects of C such as "types of object supported", we inevitably run into the proble of standards in C, or rather the lack of them. C was developed at Bell Laboratories in the late seventies, along with the UNIX operating system. The definitive tutorial introduction, programming guide and reference manual for C is "The C Programming Language" by Kernighan and Ritchie, published in 1978, and still available in the Prentice-Hall Software Series. This defines quite rigorously the syntax of C, so that it was hoped that C could remain a highly portable language.

Unfortunately, this didn't really work out, for two reasons. The first is that C itself contains very few primitives; for nearly all real activities (such as I/O) it relies on a "library" of standard functions. Unfortunately, Kernighan and Ritchie have very little to say about these, with the result that considerable variation arose in the libraries in different implementations of C. Secondly, as C gained in popularity outside its original environment (UNIX), implementers of versions of C for different systems were unable to resist adding various "improvements" to the original specification, resulting in further variations between different versions. This seems to happen inevitably to popular programming systems - the only really standard and portable systems are the ones that nobody uses!

The world is currently awaiting a new "IEEE" standard for C; this has got as far as a preliminary specification. Meanwhile, the standards are essentially determined by the major suppliers such as Lattice and Aztec; fortunately, these don't vary too greatly, and also look likely to be fairly close to the eventual new standard. In this introduction to C, we will consider facilities available in Lattice version 3.1, without worrying too much about what comes from the original Kernighan and Ritchie (K&R) standard, and what was added later. Best reference - and a classic textbook - is probably K&R itself; for information on the proposed ANSI standard see "A C Reference Manual" (2nd edition) by Harbison and Steel.

Elements of C

In order to get a feel for the "building blocks" of C, it is convenient to consider a real example - we will therefore consider (for the last time!) our simple Intuition "screen beep" program. This shows a typical structure of a C program, which can be broken down as follows.

1. The program starts with several #include statements, giving instructions to include files containing standard definitions in the compilation; this is very like using .lib to include files of standard definitions in assembler. These may be system definitions, or ones of your own. Conventionally, these are distinguished by giving the names of files containing system definitions in angle-brackets <filename>, and the names of files containing your own definitions in quotes "filename". Examples would be

```
#include <intuition/intuition.h>
#include "myfiles/mymacros.h"
```

   Note that the #include statements break the rules, in that they are not followed by semi-colons! This is because lines starting with "#" were not originally considered as lines of C, but as instructions to a "macro preprocessor" which scans the program sorting out included files and macro definitions **before** handing over to the compiler. In Lattice C, macro preprocessing is done as part of lc1.

2. The #includes are then followed by the first lines of the C program, which are typically **extern** statements telling the compiler what it needs to know about functions from other modules or from the linker library that are needed in this compilation. An example would be

```
extern APTR AllocMem();
```

   This tells the compiler that we are going to be using a function AllocMem() defined somewhere else, and that this function returns a value of type APTR, which is a general-purpose address pointer - see below.

3. Next, we may need to tell the compiler a bit about functions declared later in this module. Considering that they usually run in several passes, C compilers are generally pretty dim about function forward reference, and expect to know what sort of thing a function returns before it is first referenced. One solution to this is to build the module "bottom up", so that functions are always defined before they are referenced. Another is to declare the function in extern statements, despite the fact that they are actually not external, but declared later in **this** module - this works fine in Lattice, but isn't generally approved of. The approved thing to do is just to declare the function with a terminating semi-colon, to indicate to the

compiler that the actual definition is later in the module:

```
APTR WonderFunction();   /* defined later on */
```

The compiler will then be able to cope with references to WonderFunction() which occur before the actual definition.

3. The extern declarations and any necessary function declarations are typically followed by definitions of "global" variables which need to be accessed from a variety of different points in the program. This will cause the compiler to allocate space for the variables in DATA or BSS hunks used for initialised and uninitialised data respectively, and to output the variable name to the object file, so that other modules can access it.

   Examples are

   ```
   APTR IntuitionBase;
   ULONG Counter = 0;
   ```

   In the first case we are simply instructing the compiler to reserve space for a pointer called IntuitionBase in a BSS hunk. In the second we are reserving space for an unsigned long-word called Counter, and initialising it to zero - this will end up in a DATA hunk.

4. Following the global data comes the program itself. A C program consists of a series of "functions" which are general-purpose building-blocks, which fill the same roles as functions, procedures and subroutines in other languages. The first function of the program is conventionally called main(); this is then followed by other functions as necessary. Each function consists of a declaration giving the function name and information about what is passed to this function and what it returns, followed by a function definition enclosed in curly brackets. An example would be

   ```
   APTR GetChipMemory(amount);
   ULONG amount;
   {                          /* function defined here */

   }
   ```

   Here we are going to define a function called GetChipMemory to allocate memory in the bottom 512K of the Amiga's memory map. We are going to pass it a long-word value "amount" telling the function how much memory we want; the function is then going to return an APTR telling us the address of the memory allocated, or zero (NULL) if none was available. The fact that the value "amount" is an unsigned long-word is declared in the line "ULONG amount;", after the function declaration and before the opening curly bracket.

5. Within the function definition, the first things found are usually definitions of local ("automatic") variables; space will be reserved for these on the stack when the function is entered, and de-allocated when the function terminates, so these variables are stictly local and temporary. An example is the counter "i" in beep.c:

```
main()
{
     int i;         /* counter */
                    /* other stuff follows */
}
```

6. Following the local variables comes the actual lines of C that do something, consisting of control statements like if and for, assignments, logical operations, and calls to other functions.

Types of variable

The main types of variables in C are as follows.

Global variables are declared outside functions; the compiler reserves space for them in BSS or DATA hunks, and in the latter case sets up initial values. Global variables are "permanent" in that once space has been reserved for them, they cannot be made to go away. Information about global variables names and locations are output to the object file, so that they can be accessed by other modules

External static variables are just like global variables, but their names are not output to the object file, so they can only be accessed within the current module. This allows a number of collaborating functions to share things like buffers, which you may not want to be accessible from elsewhere in the program. An example would be

```
static char buffer[SIZE];
```

This tells the compiler to reserve space for an array of characters of size SIZE in a BSS hunk, but not to output the name "buffer" for access by separately-compiled modules.

Automatic variables are declared within function definitions; the system reserves space for them on the stack at run-time. These values are strictly temporary, and are thrown away when the function exits. Note that this is quite handy when it comes to recursion; if a function calls itself, then a new set of automatic variables will be created lower down the stack, which won't conflict with the original values. Automatic variables can be initialised on declaration: the initialisation will be performed by code at the start of the function.

Internal static variables are static variables declared within function definitions, and are private to the function in which they are declared. Static variables are not put on the stack; instead space is reserved for them in BSS or DATA hunks, so that their values are not lost when the function exits. This allows functions to maintain internal private values such as buffer pointers, which are preserved between calls to the function. Internal and external static variables are similar in the way that space is reserved for them; however the compiler allows internal static variables to be accessed only within the function in which they are defined, while external static variables can be accessed from anywhere within the source-code module in which they are defined.

Formal variables refer to function arguments, defined after the function declaration, and before the opening curly brackets. These are very similar to automatic variables in that they refer to some space on the stack, the difference being that in this case this space is grabbed off the stack by the calling function - this process is discussed in detail later.

Register variables are a special type of automatic variable - declaring them as "register" tells the compiler that they are going to be used a lot, so it should use processor registers for them if possible. Lattice allocates registers for register variables from D7 to D2 and A4 to A2. A7 is the stack pointer, A6 is used for library access and base relative addressing, while A5 is used for automatic variable access, which leaves only D0, D1, A0 and A1 free for scratch - so if you're writing assembler functions to interface with Lattice, make sure you preserve all registers except possibly these four!

Data types

The basic data types supported by Lattice C are

```
int                   -    signed 32-bit integer
long or long int      -    signed 32-bit integer
short or short int    -    signed 16-bit integer
char                  -    signed 8-bit quantity
float                 -    floating point
double or long float-      double-precision floating point
```

All of these are interpreted as signed quantities - be careful of this, as it can give rise to funny effects when comparing characters for example. If you want unsigned quantities, you can use a qualifier **unsigned** as in

```
unsigned char buffer[100];
```

The problem with these data types is that they are not guaranteed to be the same for other versions of C! For this reason, Amiga have defined their own data types in a file "exec/types.h" - these are distinguished from the Lattice types by being capitalised. If you #include this file in your compilation, you can use the Amiga data-types, which is a very good idea if you are ever likely to change versions of C. Commonly encountered Amiga data-types are

```
LONG      - signed 32-bit
ULONG     - unsigned 32-bit
WORD      - signed 16-bit
UWORD     - unsigned 16-bit
BYTE      - signed 8-bit
UBYTE     - unsigned 8-bit
STRPTR    - string pointer
APTR      - memory pointer
```

For a full list, see h-file exec/types.h.

Section 3 - Structures and Pointers

Introduction

Getting Started in C Section 2 looked at the different "storage classes" supported by C - global, external static, automatic, internal static, formal and register - and at the "arithmetic objects" supported directly by Lattice, or by using the Amiga macros LONG, ULONG, WORD, UWORD etc. This section carries on from this by starting to look at the "derived objects" built up using these elements, which include the very important notions of structures and pointers. Other derived objects are strings and arrays - these are in fact special cases of uses of pointers in C, and will be considered in section 4.

Structures and pointers

The importance of structures and pointers in C has already been mentioned in Part I; it is C's abilities to handle these in a reasonably civilised manner which is the main thing which makes it attractive as a systems-level programming language.

An example of a structure in C is an IOStdReq used in much Amiga IO - eg to communicate with the console device - as discussed in the section on "devices". The IOStdReq structure is defined in an .h file exec/io.h. You can instruct the macro pre-processor to include this in your compilation by putting

```
     #include <exec/io.h>
```

somewhere near the start of your program. The h-file contains the structure definition as follows:

```
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 of bytes transferred */
     APTR   io_Data;          /* points to data area */
     ULONG  io_Offset;        /* offset for block structured devices */
};
```

The first thing to note is that a structure definition can make use of other structures. The IOStdReq starts with a complete 20-byte Message structure, defined in exec/ports.h - the exec/io.h file itself contains the necessary logic to include exec/ports.h if this hasn't already been done, so you don't have to worry about this! This is followed by two 4-byte <u>pointers</u> to

a Device structure and to a Unit structure, which are also defined in other files which will be included automatically as necessary. Note carefully the absence or presence of "*", which indicates either an actual instance of a structure (Message structure), or of pointers to (ie addresses of) structures defined elsewhere (Device and Unit).

The initial structures and pointers are followed by some "elementary" objects, defined using the macros from exec/types.h; note the C-convention by which we use capital letters to distinguish macros. The use of macros rather than the Lattice primitives such as

```
unsigned short io_Command;
```

is to aid portability - unsigned short won't necessary give you a sixteen bit quantity on a different C compiler!

Direct use of structures

Having included exec/io.h, you can then create a global IOStdReq structure quite simply, by putting something like the following before the first function definition in your source module:

```
struct IOStdReq myrequest;
```

(This is in fact an un-brilliant thing to do, for reasons discussed later. However, on the current Amiga at any rate it **would** work okay, so we'll carry on with it for a while for the sake of illustration.)

This tells Lattice to reserve space for an IOStdReq structure in a BSS hunk, to output "myrequest" as a global symbol accessible from other modules (to avoid this, use "static struct...."), and to note that myrequest refers to this kind of structure. Elements within the structure can then be accessed using the "dot" operator, eg

```
myrequest.io_Command = CMD_READ;
```

This sets the command word in the request block to CMD_READ - this is another macro (capital letters) which is #defined in exec/io.h, and actually evaluates to 2.

Dot evaluates left to right, so it is possible to pile one on another - eg to set up the message port node type correctly, we could use

```
myrequest.io_Message.mn_Node.ln_Type = NT_MESSAGE;
```

This relies on the fact that the Message structure (defined in exec/ports.h) itself contains a Node structure mn_Node, and that the Node structure (defined in exec/nodes.h) contains a UBYTE element ln_Type. The macro NT_MESSAGE is also defined in

exec/nodes.h, and actually expands to 5.

Having set up an IOStdReq structure like this, and initialised node type, priority and address of reply port by means similar to the above, we could then attempt to use it to open the Console device as follows:

```
     if (OpenDevice("console.device",0,&myrequest,0) != 0)
          /* cope with open failed */
     else
          /* cope with open succeeded */
```

Here we are using the OpenDevice EXEC function which takes as arguments the device name, a unit number (here zero), the address of the IOStdReq block, and flags (here zero). This introduces another important C operator - "ampersand" is used to indicate that we want to pass the address of something, rather than the object itself. OpenDevice returns either zero or an error-code - here we are testing this immediately in a typically C-ish fashion using the != (not-equals) relational operator.

(In fact, C if statements behave like many BASICs, in that zero is taken as false, and non-zero as true. Thus the above could be replaced by the even-more-C-ish

```
     if (OpenDevice("console.device",0,&myrequest,0))
          /* cope with open failed */
     else
          /* cope with open succeeded */
)
```

Two further facilities regarding structures declared explicitly in the code are worth noting. The first is that it is possible to initialise a structure when declaring it - eg to set up a message node:

```
     struct Node mynode = {
          NULL,           /* *ln_Succ - pointer to next node */
          NULL,           /* *ln_Pred - pointer to previous node */
          NT_MESSAGE,     /* ln_Type - node type */
          0,              /* ln_Pri - node priority */
          "My node"       /* *ln_Name - pointer to node name */
     };
```

This is often (and quite legitimately) used to set up structures like Intuition NewWindows - see numerous examples, including CONSOLE.C elsewhere in section 3 of this publication.

Secondly, note that it is of course possible to define your own structures just as easily as using the ones provided in the .h files. Suppose you wanted to define your own structure consisting of an IOStdReq followed by an extra flags byte for your own nefarious purposes -

```
struct FunnyIoReq {
     struct IOStdReq;
     UBYTE FunnyFlags;
};
```

It is possible to combine defining a structure like this with declaring one (or more) instances of the structure, as in

```
struct FunnyIOReq {
     struct IOStdReq;
     UBYTE FunnyFlags;
} myFunnyIOReq;
```

You can also combine this with initialising the structure as above, if you want to get really carried away! Initialising global or static structures like this causes Lattice to output initialised DATA rather than uninitialised BSS hunks. (It is also possible to initialise an automatic variable or structure, but this is handled differently, by actually generating code to handle the initialisation.)

Indirect use of structures

As mentioned above, setting up an IOStdReq structure by declaring it directly in the code, then accessing it using dot, is in fact considered an unwonderful thing to do. The reason for this is that future products in the Amiga series may make use of hardware memory partitioning, so that rogue tasks can no longer burst their bounds and crash the entire machine. This will mean that structures such as messages which have to be shared between tasks have to be put in a special region of "public" memory obtained by calling AllocMem() MEMF_PUBLIC - current Amigas support this as an upward compatibility feature.

Thus if we want to be upward-compatible, we will have to obtain our memory for our IOStdReq by calling AllocMem(), then accessing it indirectly. To do this we need a pointer to the structure, which we declare as follows, probably as a global variable:

```
struct IOStdReq *myrequestpointer;
```

Having set up a pointer to a structure like this, it is now possible to access elements within the structure using another operator "->", known in-house at Ariadne as SillyArrow:

```
myrequestpointer->io_Command = CMD_READ;
myrequestpointer->io_Message.mn_Node.ln_Type = NT_MESSAGE;
```

SillyArrow also evaluates left to right, which leads to a very convenient way of dealing with pointers to pointers, eg

```
sigmask = 1<<(myrequestpointer->io_Unit->unit_MsgPort->mp_SigBit)
```

This uses myrequestpointer to pick up a pointer to a unit, from

which we pick up a pointer to a message port, from which we pick up a signal bit number; we then shift 1 left this number of times (<< is binary shift left) in order to convert this to a signal bit mask, which we could then use as an argument to the EXEC Wait() function!

Note that in C, it is often important to remember to distinguish between a pointer, and the thing being pointed to! For example, suppose we wanted to copy the IO request block that had been initialised by a call to OpenDevice, into another area accessed via another pointer "myotherpointer". We might try and do this by

```
     myotherpointer = myrequestpointer;
```

which would just copy the pointer, presumably not what we had in mind! To copy the thing being pointed at, we use the * operator again:

```
     *myotherpointer = *myrequestpointer;
```

which will copy the entire structure from one place to the other. (This ability to access a whole structure like this is in fact a special feature of Lattice, rather than standard K&R. More conventially, *pointer is used to access simple arithmetic objects using pointers - this will be discussed in the context of arrays in the next issue of Kickstart.)

Casting

If we now consider this example further and consider how we actually allocate some MEMF_PUBLIC memory for use as an IOStdReq structure, we come across an interesting difficulty. Allocating the memory is no problem, particularly if we use the C sizeof operator which returns the size of a structure in bytes:

```
     myrequestpointer
          = AllocMem(sizeof(struct IOStdReq),MEMF_CLEAR|MEMF_PUBLIC);
```

The problem is that AllocMem is usually declared to return an APTR (general purpose absolute memory pointer), eg by

```
     extern APTR AllocMem();
```

while mypointer will have been declared as a pointer to an IOStdReq structure,

```
     struct IOStdReq *mypointer;
```

C won't think much of you equating two different types of pointers, and will give you a warning message "pointers do not point to the same object". In order to avoid this, you have to perform an explicit type conversion, using an operation known as casting.

Casting is the mechanism in C which allows you to convert data types at the drop of a hat - this is yet another reason why Pascal programmers will probably dislike C, but for the rest of us ("It's all binary really innit?") it comes in very handy. A simple example of a cast would be

```
UBYTE bert;
ULONG fred;

fred = (ULONG) bert;
```

The "cast" (ULONG) tells C explicitly to convert bert to type ULONG before equating it to fred, though in fact the cast is optional here since this conversion wouldn't worry C anyway. Casting is more often used in connection with pointers, as in:

```
APTR bert;
struct funnystructure *fred;

fred = (struct funnystructure *) bert;
```

This uses casting to convert bert to a pointer to a structure "funnystructure" before equating it to fred - this would otherwise worry C, at least to the extent of issuing a 'pointers do not point' warning, since it is quite fussy about knowing what pointers are supposed to be pointing to.

Thus the "correct" way of writing the call to AllocMem in this case is as follows:

```
myrequestpointer = (struct IOStdReq *)
     AllocMem(sizeof(struct IOStdReq),MEMF_CLEAR|MEMF_PUBLIC);
```

Assuming this works okay (mypointer doesn't end up NULL), then we can proceed to initialise the message port, then call OpenDevice():

```
myrequestpointer->io_Message.mn_Node.ln_Type = NT_MESSAGE;
myrequestpointer->io_Message.mn_Node.ln_Pri = 0;
myrequestpointer->io_Message.mn_ReplyPort = myportpointer;

if (OpenDevice("console.device",0,myrequestpointer,0) != 0)
     /* cope with open failed */
else
     /* cope with open succeeded */
```

where we are assuming that "myportpointer" has already been set up as a pointer to a MessagePort structure.

In fact, it isn't necessary to do all this setting up before calling OpenDevice yourself, since this is provided by an EXEC support function CreateStdIO(address of reply port). See ROM kernel manual volume I for a listing of this function.

Summary

This section has introduced several of the most important concepts in C, and some key operators ampersand, dot, splat and SillyArrow, along with the notion of casting. A summary is as follows:

```
fred = &bert;         - set fred to the address of bert.
fred = thing.bert;    - set fred to the element bert in the
                        structure thing.
fred = *bert;         - set fred to what bert points to.
fred = thing->bert;   - set fred to the element bert in the
                        structure thing points to.
fred = (struct thing *)bert;
                      - set fred to bert, having converted it
                        to a pointer to a thing structure.
```

More on structures and pointers, especially in the context of strings and arrays, follows in the next section.

Section 4 - Arrays and Strings

Getting Started in C Section 3 looked at the crucial ideas of structures and pointers in C programs; this section goes on to consider arrays and strings, which in C are built on the more general ideas of structures and pointers.

Declaring an array

Suppose we are intending to do something or other involving disk files (say), and that we are going to need a 1000 byte buffer. One way of doing this (the one we would tend to use) would be to allocate this dynamically at run-time:

```
     buffer = AllocMem(1000,0);
```

This grabs some memory from the system, and sets up "buffer" as a pointer to it. This approach has various advantages, including that the memory allocated will be certain to start at a longword boundary (often useful for AmigaDOS), and that various options exist to specify what kind of memory you want (MEMF_FAST, MEMF_CHIP, MEMF_PUBLIC) and whether or not you want the memory zeroed for you by AllocMem (MEMF_CLEAR). The main disadvantage of doing this is that you have to remember that you have allocated this memory, and be prepared to free it when you finish - unless you are writing the kind of Wallyware where you have to reset the Amiga to exit!

Thus you may prefer to have C handle the memory allocation for you. One way of doing this is to call the Lattice standard (Unix-like) memory-allocation routines malloc(), calloc() etc - memory allocated this way is tracked by Lattice, and will be freed up when you perform a Lattice exit(). Alternatively, you don't have to use a function at all - you could just declare a global array:

```
     UBYTE     buffer[1000];
```

This essentially sets up a structure consisting of 1000 repetitions of a UBYTE, with a <u>pointer</u> "buffer" pointing at the first element in the structure. In the case of an uninitialised global array like this (declared outside the body of a function definition), the effect of this declaration will be enter a reservation for space for the array in a BSS hunk - memory will actually be allocated and the value of "buffer" resolved by the AmigaDOS scatter-loader at load time. The buffer will end up linked into the "segment list" for the program, and will be freed up automatically by AmigaDOS Exit().

Just like simple variables, global arrays can be initialised at the same time they are declared. A classic example (K&R) is a "days in month" array (not leap year) declared as follows:

```
UBYTE  days_in_month[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Arrays in functions

In the days_in_month[] example above, note that we didn't have to tell C explicitly the size of the array, since it's perfectly capable of counting the elements for itself! In general, we only have to tell C the array size when C is actually reserving space for the array. For example, if we had a function which took an array as an argument, we might declare the argment as follows:

```
cunningfunction(somearray)
UBYTE somearray[];
{
    /* some wonderful code in here */
}
```

Remembering that "somearray" is actually a pointer to the start of array storage in memory, you could just as well write

```
cunningfunction(somearray)
UBYTE *somearray;
{
    /* even more wonderful code using pointers */
}
```

Whether you continue to think of "somearray" as an array name as in the first example, or whether you work with it directly as a pointer as in the second example, is very much a matter of preference. Generally speaking, C couldn't care less - though in some cases, the pointer approach is faster.

Other array types

In the examples above, we have been considering the simplest case, consisting of a global array of UBYTEs. Static arrays (internal or external) are very similar indeed to global arrays, the only difference being that the array name is not output to the object file, so that the array can only be accessed from within a particular function (internal static), or within a particular source-code module (external static). The use of external static arrays allows a group of collaborating routines - say a group of routines concerned with disk IO - to share "private" buffers, which can't be interfered with from elsewhere in the program.

It is also possible to declare automatic arrays within functions. These are handled differently, as space is not reserved for them in DATA or BSS hunks - instead space is grabbed off the stack for

the array when the function is invoked, and given back when the function exits. Note that in standard (K&R) C it is not possible to initialise an automatic array when you declare it, so in this case you always have to tell C the array size explicitly.

Arrays of other objects

Similarly, C is not restricted to arrays of UBYTES - you can have arrays of pretty well anything you like. These can be either elementary data types such as

```
ULONG somearray[SIZE];      /* array of unsigned 32-bit quantiti
```

or derived data types, such as arrays of structures -

```
struct List listarray[SIZE];     /* array of list headers */
```

This would produce an array of List structures, each of which could have a linked list hung off it. This would be a handy thing to have in a database or similar application, where you wanted to hash on a key, then search through a hash-chain of elements with the same hash-value - ie to use the same technique as AmigaDOS uses to find a named file or directory. Having declared an array of list headers above, you could then get at the right list by

```
thislistpointer = &listarray[hashvalue%SIZE];
```

(% is integer modulus in C).

As well as arrays of structures, you can of course have arrays of pointers - perhaps better known to you or me as an address table. For example, if we were going to write a utility to print a list of waiting tasks in order of task priority, then we would probably declare an arrays as follows:

```
struct Task *taskpointers[SIZE];
```

We could then set up this array by calling Disable() then bunny-hopping through Exec's waiting list (the list header is in the positive offsets from ExecBase); we could then sort the list on priority of task pointed at, and finally print out the task names in the right order.

Finally, we can have arrays of arrays. The declaration

```
UBYTE     thing[12][5];
```

declares an array of 12 objects, each of which is an array of 5 UBYTES - in other words a two dimensional array. Arrays of three and more dimensions follow just as naturally - quite neat really, isn't it?

Accessing arrays - pointer arithmetic

Suppose we have an array of longwords, declared

```
ULONG    thing[SIZE];
```

Then we can access the i'th element in this array by

```
element = thing[i];
```

(Note that array indexing starts from zero, so i will range from 0 to SIZE-1.) Alternatively, we can make use of the fact that "thing" is actually the address of the array in memory, so we can just as well write this

```
element = *(thing+i);
```

The reason that this works is because pointer arithmetic in C is scaled according to the size of the object pointed to. Thus if we are trying to access the fourth element in the array, then (thing+i) will actually cause sixteen to be added on to thing, because C knows that thing is an array of longwords (size 4). This will result in an offset from the base of the array of sixteen bytes, which is four longwords - exactly what we wanted.

Other pointer arithmetic works in the same way. For example, suppose that we have a pointer into our array set up by

```
pointer = thing;
```

This is exactly equivalent to

```
pointer = &thing[0];
```

i.e. it points at the first element in the array. Now

```
pointer++;
```

will increment pointer by the size of object pointed at - ie by four bytes in this case - so that it points to the next element in the array. Note a small difference between pointers and arrays here, in that a pointer is a variable while an array name is a constant. Thus if we want to do this sort of thing we have to set

```
pointer = thing;
pointer++;
```

as an attempt to perform

```
thing++;
```

would result in an error.

Similarly

```
pointer--;
```

will decrement pointer by four bytes, to point at the previous element. If we have two pointers into the array, then pointer subtraction will be scaled in the same way -

```
thing = pointer1-pointer2
```

will subtract the two pointers then scale the result down, to return the number of elements between the two pointers. We can also compare two pointers in a similar way -

```
if (pointer1 < pointer2) then {

     /* whatever */

}
```

However these are the only operations which are legal - pointers can be incremented or decremented, they can have integers added or subtracted from them, and they can be subtracted or compared. Other operations, such as attempting to add two pointers or multiply by a constant, are illegal and result in a compiler error. Note also that operations such as pointer subtraction can only be scaled sensibly if the pointers point to the same object, eg to elements in the same array - attempts to perform pointer arithmetic on pointers to different objects will result in a fatal compiler error.

All there is to know about strings in C

In the same way that arrays are really special cases of pointers and structures in C, so strings are special cases of arrays - a C string is simply a null-terminated array of characters. Thus declaration of a literal string

```
fred = "here is a string";
```

will actually cause "here is a string" followed by a null to be output as an array of characters in the data hunk from the compilation; fred will be a pointer to this array, ie the address of "h". Note that

```
fred = 'h';
```

and

```
fred = "h";
```

are totally different in C - the first sets just sets fred to the ASCII value for 'h', while the second outputs an "h" followed by a null in a data hunk, and sets fred to the address of it.

In terms of string manipulation, what C provides in this respect is, er, nothing. The Lattice library lc.lib provides standard functions to perform string concatenation, string-chopping, comparision etc. The fact that you have to call library functions to perform string operations is another weakness of C viewed as a high-level language - but then it isn't really a high-level language anyway!

Section 5 - Getting Finished in C

This last section of Getting Started in C looks at flow control in C - ie at if...else constructions, at various forms of loop, and at how control is passed forwards and backwards between functions.

If...else

The basic "if" construction in C is quite straightforward:

```
     if (expression) statement;

OR   if (expression) statement1; else statement2;
```

In the first case, statement will only be executed if expression is non-zero (true). In the second case, statement1 will be executed if expression is non-zero, otherwise statement2 will be executed. statement can be a single line of C:

```
if (x == 42) printf("Meaning of life discovered\n");
```

Here we are using the equality operator "==" (NOT to be confused with the assignment operator "=") which returns TRUE (1) if two things are equal, and FALSE (0) otherwise.

Alternatively, if can be followed by a series of statements surrounded by braces:

```
if (handle = Open("Thing",MODE_OLDFILE)) {
     printf("File Thing opened okay\n");
     readfile(handle);
} else {
     printf("Couldn't open file Thing\n");
     exit(20);
}
```

Here we are attempting to open a file "Thing"; it successful we read it in using a function (assumed defined somewhere else) called readfile(), otherwise we exit using the Lattice standard function exit(). The AmigaDOS Open() function returns either a "handle" describing the file, or else returns zero if it fails, so we can set up variable handle AND test it for being non-zero in one operation as shown above. Again, notice the difference between = and ==!

Other logical tests

It is frequently necessary to set a variable to one of two possible values depending on the result of a logical test. For example, if we have just got information using AmigaDOS Examine(and want to remember whether what we were looking at was a file or a directory, we might use

```
          if (fib->fib_DirEntryType > 0)
               type = DIRECTORY;
          else
               type = FILE;
```

(Here DIRECTORY and FILE are two macros we have defined somewhere for our own use.) C provides a short-cut for doing this sort of thing, in the form of a "conditional expression" as follows:

```
     type = (fib->fib_DirEntryType > 0) ? DIRECTORY : FILE ;
```

This works out the expression before the "?" then sets type to DIRECTORY or FILE, depending on whether the expression is true (non-zero) or false respectively.

Another shortcut is provided when it is necessary to take a number of possible courses of action depending on different possible integer values, eg in the case of different possible messages received from an Intuition IDCMP:

```
     if (message->Class == MOUSEBUTTONS) {
          /* cope with mouse button pressed */
     }
     if (message->Class == MENUPICK) {
          /* cope with menu selection */
     }
     if (message->Class == CLOSEWINDOW) {
          /* window closed - exit */
     }
          etc.
```

This can be handled much more neatly as follows:

```
     switch(message->Class) {

          case(MOUSEBUTTONS):
               /* cope with mouse button pressed */
          break;

          case(MENUPICK):
               /* cope with menu selection */
          break;

          case(CLOSEWINDOW):
               /* window closed - exit */
          break;

          default:
               /* unrecognised message - panic! */
     }
```

Note the break statements which are necessary to get you out of the switch braces and onto the next statement; without this each case will fall into the next one. Note also the default:, which

specifies what to do if none of the cases specified are satisfied - in this case we would be getting an unrecognised message from Intuition, which would almost certainly indicate dire trouble!

Looping

All looping in C is a variation on "while", with the test either at the start or at the end of the loop:

```
     while (expression) statement;           (test at start)
 OR  do statement; while (expression);       (test at end)
```

Note the terminating semi-colon in the second case (do-while). Again, statement can be either a single line of C terminated by a semi-colon, or a series of lines enclosed in braces:

```
while (ExNext(lock,fib)) {
     if (fib->fib_DirEntryType > 0)
          printf("Directory");
     else
          printf("File");
     printf(" %s\n",fib->fib_FileName);
     }
```

Here we are printing out an AmigaDOS directory listing, using a function ExNext which fills in the next entry in a "file information block", and returns zero when there are no more entries or if it hits an error.

A "for" loop in C is really a variant on "while", with the test at the top of the loop:

```
for (start-expression; while-condition; loop-expression)
     statement;
```

The C equivalent of FOR X=1 TO 10:PRINT"WOMBAT":NEXT is therefore

```
for (x=1 ; x<=10; x++) printf("wombat\n");
```

However, the C "for" is much more powerful than BASIC FOR...NEXT, the trick being the loop-expression which specifies what to do between one iteration and the next. For example, to "bunny hop" through a linked list, we can use a loop-expression which picks up the pointer to the next node in the list from the current node:

```
for (node = list->lh_Head; next = node->ln_Succ; node = next) {
     /* process list in here */
}
```

Here the while-condition is (next = node->ln_Succ), meaning keep looping until the pointer to the next node is zero.

It is possible to have more than one start-expression, while-expression or loop-expression separated by commas; it is also possible for any or all of these expressions to be be null:

```
     for (;;) {
          /* keep looping forever */
     }
```

A frequently encountered trick in C is to use a for-loop whose body is null - eg to copy a string from one place to another

```
     for (i = 0;destn[i] = source[i];i++);
```

Here the while-condition is being used both to copy a character and test if it is null; the actual body of the loop does nothing, as indicated by the terminating semi-colon. This is a point to watch - it is quite easy to create a null loop by accident, by putting a semi-colon a the end of a "for" statement when you don't mean to!

Getting out of loops

Loops in C don't use the stack, so it is usually quite safe to break out of them. The "clean" way to do this is using the keyword break - eg we might want to keep looping until we get an Intuition CLOSEWINDOW message:

```
     FOREVER {           /* Intuition macro - expands to for (;;) */

          WaitPort(OurWindow->UserPort);     /* wait for message */
          message = GetMsg(OurWindow->UserPort);
                       /* (should be checking for more than one) */

          class = message->Class;
          if (class == CLOSEWINDOW) break;

          switch(class) {
               /* handle other message types */
          }
     }
     /* processing continues here after break */
```

Another way out is to use (shudder) goto:

```
     if (!(buffer = AllocMem(1000,0)) goto nomemory;
               /*** buffer allocated okay - continue ***/

     nomemory:
               /*** run out of memory - clean up and exit ***/
```

Our own opinion is that this sort of thing is perfectly okay for handling errors - its use for other purposes should be avoided however.

Flow control using functions

The passing of control between functions in C is illustrated in listing 1, which shows a program to add two numbers using a (totally unnecessary) function sum(a,b). Looking first at function main() - starting at offset 0010 in the object module disassembly - what happens is as follows:

1. First of all we set up two global variables "first" and "second". The compiler has put these in section 02 (BSS hunk), at offsets 0 and 4.

2. Now we want to call function sum(), passing it the current values of "first" and "second" as parameters. To do this, we take the current value of "second" (already in D0), and push it to the stack, followed by the current value of "first". We now call sum() by BSR 0 - this takes us to offset zero in the object module disassembly.

3. The first thing done in sum() is to reserve space for a temporary ("automatic") variable c. This is done by LINK A5,FFFC, which grabs another four bytes off the stack, and puts a pointer to them in A5.

4. sum() can now get at its formal variables a and b, and at its automatic variable c, by looking at what has been pushed to the stack. It does this by using A5 with offsets zero (c), 8 (a) and 12 (b).

5. When sum() has finished, it first of all cleans up the space it grabbed off the stack by UNLK A5; it then returns a value by passing it back in D0.

6. When main() is returned to, it first of all cleans up the values it pushed to the stack by ADDQ.L #8,A7; it then sorts out "first" by moving D0 to the appropriate offset in hunk 2. This completes the story.

Note that the values of the formal variables a and b, and of the automatic variable c, are all strictly temporary - these variables consist of some space at the top of the stack which is thrown away when the function has finished either by the function itself (c), or by the function which called it (a and b). This scheme is known as "passing by value" - the function sum() is given the current values of first and second at the top of the stack, but has no access to the variables themselves where they are stored in hunk 2, and can't affect their values.

You have to be a bit careful about this in C programs. For example, you might think it was possible to swap the values of two variables "first" and "second" by calling swap(first, second) where function swap() is defined as follows:

```
void swap(a,b)
int a,b;
{
     int c;
     c=a; a=b; b=c;
}
```

However, all this will do is to fiddle around with temporary values at the top of the stack which will then be thrown away. The way to get round this - ie to provide a facility for "passing by reference" in which a swap() function can affect the current values of its arguments is to pass it the address of its arguments, then let it access them indirectly. Thus you would call swap(&first, &second) and define swap as follows:

```
void swap(a,b)
int *a,*b;
{
     int c;
     c = *a; *a = *b; *b = c;
}
```

An OMD for this is given in listing 2 - checking though what the compiler does in this case and that it works out correctly is left as an exercise for the reader!

A final point

Note that while C generally checks the type of value returned by a function very carefully, it doesn't usually make much of a fuss about values passed to it!. Thus if you called a function sum(a,b) with two floating point parameters, but had defined the function to expect integers, then sum() would read values off the top of the stack as if it had been passed integers, resulting in a garbage result. In Lattice it is possible to protect yourself against this by explicitly stating argument types expected by functions, eg by

```
extern sum(int,int);
```

though this is not compulsory. In Lattice it is also permitted to pass complete structures to functions and to return them from functions - however this is unusual, and Lattice will generally give a warning message to check that this is really what you had in mind!

What to do now

Read Kernighan and Ritchie.

```
/* Trivial example for Kickstart */

int first,second;                       /* two globals */

/* unnecessary function to add two numbers */

int sum(a,b)
int a,b;
{
   int c;                               /* auto variable - temp storage */

   c = a+b;
   return(c);
}

void main()
{
   first = 2;
   second = 4;
   first = sum(first,second);
}
```

```
LATTICE OBJECT MODULE DISASSEMBLER V2.00

Amiga Object File Loader V1.00
68000 Instruction Set

EXTERNAL DEFINITIONS

_sum 0000-00    _main 0010-00    _first 0000-02    _second 0004-02

SECTION 00 "add.o" 00000034 BYTES
0000 4E55FFFC                     LINK       A5,FFFC
0004 202D0008                     MOVE.L     0008(A5),D0
0008 D0AD000C                     ADD.L      000C(A5),D0
000C 4E5D                         UNLK       A5
000E 4E75                         RTS
0010 7002                         MOVEQ      #02,D0
0012 23C0 00000000-02             MOVE.L     D0,02.00000000
0018 7004                         MOVEQ      #04,D0
001A 23C0 00000004-02             MOVE.L     D0,02.00000004
0020 2F00                         MOVE.L     D0,-(A7)
0022 2F39 00000000-02             MOVE.L     02.00000000,-(A7)
0028 61D6                         BSR        00000000
002A 508F                         ADDQ.L     #8,A7
002C 23C0 00000000-02             MOVE.L     D0,02.00000000
0032 4E75                         RTS

SECTION 01 " " 00000000 BYTES

SECTION 02 " " 00000008 BYTES
```

Listing 1 - sum function

```
/* Swap example - debugged version */

int first,second;                  /* two globals */

/* function to swap two numbers */

void swap(a,b)
int *a,*b;
{
   int c;                          /* auto variable - temp storage */

   c  = *a;
   *a = *b;
   *b = c;
}

void main()
{
   first = 2;
   second = 4;
   swap(&first,&second);
}
```

```
LATTICE OBJECT MODULE DISASSEMBLER V2.00

Amiga Object File Loader V1.00
68000 Instruction Set

EXTERNAL DEFINITIONS

_swap 0000-00    _main 001A-00    _first 0000-02    _second 0004-02

SECTION 00 "swap.o" 0000003C BYTES
0000 4E55FFFC                     LINK       A5,FFFC
0004 206D0008                     MOVEA.L    0008(A5),A0
0008 2010                         MOVE.L     (A0),D0
000A 206D000C                     MOVEA.L    000C(A5),A0
000E 226D0008                     MOVEA.L    0008(A5),A1
0012 2290                         MOVE.L     (A0),(A1)
0014 2080                         MOVE.L     D0,(A0)
0016 4E5D                         UNLK       A5
0018 4E75                         RTS
001A 7002                         MOVEQ      #02,D0
001C 23C0 00000000-02             MOVE.L     D0,02.00000000
0022 7004                         MOVEQ      #04,D0
0024 23C0 00000004-02             MOVE.L     D0,02.00000004
002A 4879 00000004-02             PEA        02.00000004
0030 4879 00000000-02             PEA        02.00000000
0036 61C8                         BSR        00000000
0038 508F                         ADDQ.L     #8,A7
003A 4E75                         RTS

SECTION 01 " " 00000000 BYTES

SECTION 02 " " 00000008 BYTES
```

Listing 2 - swap function

Appendix - 1.2 libraries summary

Following is a summary of all routines in all libraries in version 1.2, originally prepared by Mike Bolley for Kickstart Issue 5. This was based on Kickstart 33.166 and Workbench 33.43, a bit earlier than the final 1.2 release versions, which were Kickstart 33.180 and Workbench 33.47. We don't expect library contents to have changed at all in the release versions however.

Note that by the time of Kickstart 33.166, clist.library - the Amiga ROM string-handling - had already gone missing. At the time its fate seemed uncertain, so it was included in Mike's summary - in the end however it was left out of the release 1.2 ROM (Kickstart 33.180), presumably for reasons of ROM-space, and on the grounds that no-one was using it.

The objective of this summary is to give an idea of "what's where in the Amiga", and to provide a quick reference for finding routines when you need them. For full routine descriptions, see the official documentation, as provided in the 1.2 autodocs.

System libraries

Amiga system libraries are provided either on a KickStart disk (KICK) and loaded into "ROM" on boot-up, or on disk in the libs: directory, in which case they are loaded into memory as required.

This document describes the system libraries provided on Release 1.2 dated 01-Oct-86, comprising KickStart 33.166 and WorkBench 33.43. Note that the clist.library does not seem to be provided in this release, but presumably will be restored in the final 1.2 release.

Details of the library version and revision numbers, etc are given for each library, together with a list of all library functions, giving their offsets from the library base and brief descriptions.

For full descriptions of library functions, see the official documentation, provided on the 1.2 ReadMe disk.

Library name	Description	Where
clist.library	Character-list handling functions	????
diskfont.library	Disk-resident text font handling	libs:
dos.library	AmigaDOS functions	KICK
exec.library	General system functions	KICK
expansion.library	Functions for handling system expansion	KICK
graphics.library	Graphics functions, including text	KICK
icon.library	Icon handling and related functions	libs:
info.library	Used by WorkBench 'info'	libs:
intuition.library	Intuition user-interface handling	KICK
layers.library	Layer-handling functions	KICK
mathffp.library	Fast Floating Point (FFP) arithmetic	KICK
mathieeedoubbas.library	IEEE double precision format arithmetic	libs:
mathtrans.library	Fast Floating Point Transcendental Funcs	libs:
ramlib.library	RAM-disk handling functions	KICK
translator.library	Translate text to phoneme stream	libs:
version.library	Version number, & little else!	libs:

clist.library: Character-list handling functions

Name: "clist.library"
Version: ??
Revision: ??
IdString: ??
NegSize: ??
PosSize: ??

Where: ?? where indeed ??

Base Name: ClistBase

Function name	Description
FF64 ConcatCList	concatenate two character-lists (clists)
FF6A SubCList	copy a substring from a clist
FF70 CopyCList	copy a clist to a new clist
FF76 SplitCList	split a clist
FF7C PeekCLMark	peek at the byte in the clist at the mark
FF82 IncrCLMark	increment a clist mark to the next position
FF88 MarkCList	mark a position in a clist
FF8E GetCLBuf	convert a characater-list to contiguous data
FF94 PutCLBuf	convert contiguous data into a character-list
FF9A UnPutCLWord	get a word from the end of a character-list
FFA0 UnGetCLWord	add a word to the beginning of a character-list
FFA6 GetCLWord	get a word from the beginning of a character-list
FFAC PutCLWord	add a word to the end of a character-list
FFB2 UnPutCLChar	get a byte from the end of a character-list
FFB8 UnGetCLChar	add a byte to the beginning of a character-list
FFBE GetCLChar	get a byte from the beginning of a character-list
FFC4 PutCLChar	add a byte to the end of a character-list
FFCA SizeCList	get the number of bytes in a character-list
FFD0 FlushCList	clear a character-list
FFD6 FreeCList	free a clist
FFDC AllocCList	allocate and initialize a clist
FFE2 InitCLPool	initialize a clist pool

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

diskfont.library: Functions for handling disk-resident text fonts

Name: "diskfont.library"
Version: 33
Revision: 16
IdString: NONE
NegSize: $0024
PosSize: $0038

Where: Disk

Base Name: DiskfontBase

Function name	Description
FFDC AvailFonts	build an array of all fonts in memory / on disk
FFE2 OpenDiskFont	load and get a pointer to a disk font

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

dos.library: AmigaDOS functions

Name: "dos.library"
Version: 33
Revision: 124
IdString: "dos 33.110 (11 Sep 1986)"
NegSize: $00DE
PosSize: $007C

Where: KickStart "ROM"

Base Name: DOSBase

Function name	Description
FF22 Execute	execute a CLI command
FF28 IsInteractive	discover whether a file is a virtual terminal
FF2E ParentDir	obtain the parent of a directory or file
FF34 WaitForChar	determine if chars arrive within a time limit
FF3A Delay	delay a process for a specified time
FF40 DateStamp	obtain the date and time in internal format
FF46 SetProtection	set protection for a file or directory
FF4C SetComment	change a file's comment string
FF52 DeviceProc	return the process ID of specific I/O handler
FF58 QueuePacket	send a packet to another process
FF5E GetPacket	get, or wait for, a packet
FF64 UnLoadSeg	unload a segment previously loaded by LoadSeg()
FF6A LoadSeg	load a load module into memory
FF70 Exit	exit from a program
FF76 CreateProc	create a new process
FF7C IoErr	return extra information from the system
FF82 CurrentDir	make a dir associated with a lock the working dir
FF88 CreateDir	create a new directory
FF8E Info	returns information about the disk
FF94 ExNext	examine the next entry in a directory
FF9A Examine	examine a directory or file associated with a lock
FFA0 DupLock	duplicate a lock
FFA6 UnLock	release a lock on a directory or file
FFAC Lock	lock a directory or file
FFB2 Rename	rename a directory or file
FFB8 DeleteFile	delete a file or directory
FFBE Seek	find and point at the logical position in a file
FFC4 Output	identify the program's initial output file handle
FFCA Input	identify the program's initial input file handle
FFD0 Write	write bytes of data to a file
FFD6 Read	read bytes of data from a file
FFDC Close	close a file for input or output
FFE2 Open	open a file for input or output

intuition.library: Intuition user-interface handling functions

Name:	"intuition.library"
Version:	33
Revision:	702
IdString:	NONE
NegSize:	$01D4
PosSize:	$0564
Where:	KickStart "ROM"
Base Name:	IntuitionBase

Function name	Description
FE2C NewModifyProp	ModifyProp(), but with selective refresh
FE32 ActivateGadget	attempt to activate a string gadget
FE38 RefreshWindowFrame	ask Intuition to redraw your window border/gadgets
FE3E ActivateWindow **(A0)**	activate an Intuition window
FE44 RemoveGList	removes a sublist of gadgets from a window
FE4A AddGList	add a linked list of gadgets to a window/requester
FE50 RefreshGList	refresh (redraw) a chosen number of gadgets
FE56 GetScreenData	get a copy of a screen data structure
FE5C UnlockIBase	surrender an Intuition lock obtained by LockIBase
FE62 LockIBase	Intuition user's access to Intuition locking
FE68 FreeRemember	free memory allocated by calls to AllocRemember()
FE6E AlohaWorkbench	WorkBench becoming active/shutting down
FE74 AllocRemember	AllocMem() & create link node to make FreeMem easy
FE7A RethinkDisplay	grand manipulator of the entire Intuition display
FE80 RemakeDisplay	remake the entire Intuition display
FE86 MakeScreen	Intuition-integrated MakeVPort() of custom screen
FE8C FreeSysRequest	frees resources used by call to BuildSysRequest()
FE92 EndRefresh	ends the optimized refresh state of the window
FE98 BuildSysRequest	build and display a system requester
FE9E BeginRefresh	sets up a Window for optimized refreshing
FEA4 AutoRequest	automatically build & get response from Requester
FEAA WBenchToFront	brings WorkBench Screen in front of all screens
FEB0 WBenchToBack	sends WorkBench Screen to back of all screens
FEB6 IntuiTextLength	returns the length (pixel-width) of an IntuiText
FEBC SetPrefs	set Intuition Preferences
FEC2 WindowLimits	set the minimum & maximum limits of the Window
FEC8 WindowToFront	ask Intuition to bring this Window to the front
FECE WindowToBack	ask Intuition to send this Window to the back
FED4 ViewPortAddress	returns the address of a Window's ViewPort
FEDA ViewAddress	returns the address of the Intuition View struct
FEE0 SizeWindow	ask Intuition to size a Window
FEE6 ShowTitle	set the screen title bar display mode
FEEC SetWindowTitles	set the Window's titles for both Window & Screen
FEF2 SetPointer	sets a Window with its own pointer
FEF8 SetMenuStrip	attaches the Menu strip to the Window
FEFE SetDMRequest	sets the DMRequest of the Window
FF04 ScreenToFront	brings specified Screen to the front of display
FF0A ScreenToBack	sends specified Screen to back of the display
FF10 Request	activates a Requester
FF16 ReportMouse	tells Intuition whether to report mouse movement
FF1C RemoveGadget	removes a Gadget from a Window
FF22 RefreshGadgets	refresh (redraw) the Gadget display
FF28 PrintIText	prints the text according to IntuiText argument
FF2E OpenWorkBench	opens the WorkBench Screen
FF34 OpenWindow	opens an Intuition Window
FF3A OpenScreen	opens an Intuition Screen
FF40 OnMenu	enables the given menu or menu item
FF46 OnGadget	enables the specified Gadget
FF4C OffMenu	disables the given menu or menu item
FF52 OffGadget	disables the specified Gadget
FF58 MoveWindow	ask Intuition to move a Window
FF5E MoveScreen	attempts to move the Screen by given increments
FF64 ModifyProp	modify current parameters of a Proportional Gadget
FF6A ModifyIDCMP	modify the state of the Window's IDCMPFlags
FF70 ItemAddress	returns the address of the specified MenuItem
FF76 InitRequester	initializes a Requester structure
FF7C GetPrefs	get current setting of the Intuition Preferences
FF82 GetDefPrefs	get a copy of Intuition default Preferences
FF88 EndRequest	ends the Request and resets the Window
FF8E DrawImage	draws the specified Image into the RastPort
FF94 DrawBorder	draws the specified Border into the RastPort
FF9A DoubleClick	test two time values for double-click timing
FFA0 DisplayBeep	flashes the video display
FFA6 DisplayAlert	create the display of an Alert message
FFAC CurrentTime	get the current time values
FFB2 CloseWorkBench	closes the WorkBench Screen
FFB8 CloseWindow	closes an Intuition Window
FFBE CloseScreen	closes an Intuition Screen
FFC4 ClearPointer	clears the mouse pointer definition from a Window
FFCA ClearMenuStrip	clears (detaches) the Menu strip from a Window
FFD0 ClearDMRequest	clears (detaches) the DMRequest of the Window
FFD6 AddGadget	add a Gadget to Gadget list of a Window or Screen
FFDC Intuition	[Intuition input event handler]
FFE2 OpenIntuition	[not documented - internal use only]

layers.library: Layer-handling functions

Name:	"layers.library"
Version:	33
Revision:	31
IdString:	NONE
NegSize:	$00AE
PosSize:	$002A

Where: KickStart "ROM"

Base Name: LayersBase

	Function name	Description
FF52	InstallClipRegion	install clip region in layer
FF58	MoveLayerInFrontOf	put layer in front of another layer
FF5E	ThinLayerInfo	convert 1.1 LayerInfo to 1.0 LayerInfo (OBSOLETE)
FF64	FattenLayerInfo	convert 1.0 LayerInfo to 1.1 LayerInfo (OBSOLETE)
FF6A	DisposeLayerInfo	return all memory for LayerInfo to memory pool
FF70	NewLayerInfo	allocate & initialize full LayerInfo structure
FF76	UnlockLayerInfo	unlock a LayerInfo structure
FF7C	WhichLayer	which layer is this point in?
FF82	SwapBitsRastPortClipRect	swap bits between bitmap & obscured ClipRect
FF88	LockLayerInfo	lock a LayerInfo structure
FF8E	UnlockLayers	unlock all layers from graphics output
FF94	LockLayers	lock all layers from graphics output
FF9A	UnlockLayer	unlock layer & allow graphics routines to use it
FFA0	LockLayer	lock layer to make changes to ClipRects
FFA6	DeleteLayer	delete layer from layer list
FFAC	EndUpdate	remove damage list & restore state of layer
FFB2	BeginUpdate	prepare to repair damaged layer
FFB8	ScrollLayer	scroll around in a layer
FFBE	SizeLayer	change the size of a non-backdrop layer
FFC4	MoveLayer	move layer to a new position in BitMap
FFCA	BehindLayer	put layer behind other layers
FFD0	UpfrontLayer	put layer in front of all other layers
FFD6	CreateBehindLayer	create a new layer behind all existing layers
FFDC	CreateUpfrontLayer	create a new layer on top of existing layers
FFE2	InitLayers	initialize LayerInfo structure (OBSOLETE)

graphics.library: Graphics functions, including text

Name: "graphics.library"
Version: 33
Revision: 89
IdString: NONE
NegSize: $028E
PosSize: $00F2

Where: KickStart "ROM"

Base Name: GfxBase

Function name	Description
FD72 AttemptLockLayerRom	attempt to lock layer structure by ROM code
FD78 GraphicsReserved2	[reserved]
FD7E GraphicsReserved1	[reserved]
FD84 BltMaskBitMapRastPort	blit from bitmap to rastport with masking
FD8A SetRGB4CM	set one color register for this ColorMap
FD90 AndRegionRegion	perform AND of one region with second region
FD96 XorRegionRegion	perform XOR of one region with second region
FD9C OrRegionRegion	perform OR of one region with second region
FDA2 BltBitMapRastPort	blit from source bitmap to dest rastport
FDA8 FreeGBuffers	deallocate memory obtained by GetGBuffers()
FDAE UCopperListInit	[not documented]
FDB4 ScrollVPort	reinterpret RasInfo information in ViewPort
FDBA GetRGB4	inquire value of entry in ColorMap
FDC0 FreeColorMap	free ColorMap structure
FDC6 GetColorMap	allocate and initialize a ColorMap
FDCC FreeCprList	deallocate hardware copper list
FDD2 XorRectRegion	perform XOR of rectangle with region
FDD8 ClipBlit	calls BltBitMap() after accounting for windows
FDDE FreeCopList	deallocate intermediate copper list
FDE4 FreeVPortCopLists	deallocate intermediate copper lists from ViewPort
FDEA DisposeRegion	deallocate all space for this region
FDF0 ClearRegion	remove all rectangles from region
FDF6 ClearRectRegion	perform CLEAR operation of rectangle with region
FDFC NewRegion	get a clear region
FE02 OrRectRegion	perform OR operation of rectangle with region
FE08 AndRectRegion	perform AND operation of rectangle with region
FE0E FreeRaster	free space for a bitplane
FE14 AllocRaster	allocate space for a bitplane
FE1A RemFont	remove a font from the system list
FE20 AddFont	add a font to the system list
FE26 AskFont	get the text attributes of the current font
FE2C InitTmpRas	init area of local memory for areafill, etc
FE32 DisownBlitter	return blitter to free state
FE38 OwnBlitter	get the blitter for private usage
FE3E CopySBitMap	synchronize layer window with SuperBitMap
FE44 SyncSBitMap	synchronize Super BitMap with layer
FE4A UnlockLayerRom	unlock layer structure by ROM code
FE50 LockLayerRom	lock layer structure by ROM code
FE56 MoveSprite	move sprite to point relative to top of ViewPort
FE5C ChangeSprite	change a sprite's image pointer
FE62 FreeSprite	return sprite for use by others
FE68 GetSprite	attempt to get a sprite for simple sprite manager
FE6E WaitBOVP	wait until vertical beam at bottom of ViewPort
FE74 ScrollRaster	push bits in rectangle in raster around by dx,dy
FE7A InitBitMap	initialize BitMap structure with input values
FE80 VBeamPos	get vertical beam position at this instant
FE86 CWait	append copper wait instruction to user copper list
FE8C CMove	append copper move instruction to user copper list
FE92 CBump	bump pointer to next instruction in copper list
FE98 InitView	initialize View structure
FE9E SetDrMd	set drawing mode
FEA4 SetBPen	set secondary pen
FEAA SetAPen	set primary pen
FEB0 PolyDraw	draw lines from a table of (x,y) values
FEB6 Flood	flood rastport like areafill
FEBC WritePixel	change pen num of one specific pixel in a RastPort
FEC2 ReadPixel	read pen number at specific location in a RastPort
FEC8 BltPattern	blit through a mask using standard areafill
FECE RectFill	fill rectangular area
FED4 BltClear	clear a block of memory words to zero
FEDA QBSBlit	synchronize blitter request with the video beam
FEE0 SetRGB4	set one color regikster for this ViewPort
FEE6 InitArea	initialize vector collection matrix
FEEC QBlit	queue up a request for blitter usage
FEF2 WaitTOF	wait for the top of the next video frame
FEF8 AreaEnd	process table of vectors and produce areafill
FEFE AreaDraw	add a point to a list of endpoints for areafill
FF04 AreaMove	define new starting point for new shape
FF0A Draw	draw line from current pen posn to specified posn
FF10 Move	move graphics pen position
FF16 SetRast	set an entire drawing area to a specified colour
FF1C WaitBlit	wait for the blitter to be finished
FF22 LoadView	use a copper instruction list to create display
FF28 MakeVPort	generate display copper list
FF2E MrgCop	merge together coprocessor instruction lists

graphics.library (contd)

	Function name	Description
FF34	InitVPort	initialize ViewPort structure
FF3A	InitRastPort	initialize RastPort structure
FF40	LoadRGB4	load RBG color values from table
FF46	AreaEllipse	add an ellipse to areainfo list for areafill
FF4C	DrawEllipse	draw an ellipse
FF52	InitGMasks	initialize all of the masks of an AnimOb
FF58	GetGBuffers	attempt to allocate all buffers of an AnimOb
FF5E	Animate	processes every AnimOb in curent animation list
FF64	AddAnimOb	add an AnimOb to the linked list of AnimObs
FF6A	SortGList	sort the current gel list, ordering y,x coords
FF70	SetCollision	set pointer to a user collision routine
FF76	RemVSprite	remove a VSprite from the current gel list
FF7C	RemIBob	immediately remove a Bob from gel list & RastPort
FF82	InitMasks	init the BorderLine & CollMask masks of a VSprite
FF88	InitGels	initialize a gel list
FF8E	DrawGList	process gel list, queueing VSprites, drawing Bobs
FF94	DoCollision	test every gel in gel list for collisions
FF9A	AddVSprite	add a VSprite to the current gel list
FFA0	AddBob	add a Bob to the current gel list
FFA6	SetSoftStyle	set the soft style of the current font
FFAC	AskSoftStyle	get the soft style bits of the current font
FFB2	CloseFont	release a pointer to a system font
FFB8	OpenFont	get a pointer to a system font
FFBE	SetFont	set the text font & attributes in a RastPort
FFC4	Text	write text characters (no formatting)
FFCA	TextLength	determine raster length of text data
FFD0	ClearScreen	clear from current position to end of RastPort
FFD6	ClearEOL	clear from current position to end of line
FFDC	BltTemplate	cookie cut a shape in a rectangle to a RastPort
FFE2	BltBitMap	move a rectangular region of bits in a BitMap

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

icon.library: Icon handling and related functions

Name: "icon.library"
Version: 33
Revision: 127
IdString: "icon 33.127 (22 Jul 1986)"
NegSize: $006C
PosSize: $002E

Where: Disk

Base Name: IconBase

	Function name	Description
FF94	BumpRevision	reformat a name for a second copy
FF9A	MatchToolValue	check a tool type variable for a particular value
FFA0	FindToolType	find the value of a tool type variable
FFA6	FreeDiskObject	free all memory in a Workbench disk object
FFAC	PutDiskObject	write out a DiskObject to disk
FFB2	GetDiskObject	read in a Workbench disk object
FFB8	AddFreeList	add memory to the free list
FFBE	AllocWBObject	allocate a Workbench object
FFC4	FreeWBObject	free all memory in a Workbench object
FFCA	FreeFreeList	free all memory in a free list
FFD0	PutIcon	write out a DiskOject to disk (as PutDiskObject)
FFD6	GetIcon	read in a DiskObject structure from disk
FFDC	PutWBObject	write out a Workbench object
FFE2	GetWBObject	read in a Workbench object

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

info.library: Function used by WorkBench 'info'

Name: "info.library"
Version: 33
Revision: 0
IdString: "info V1.2"
NegSize: $001E
PosSize: $0026

Where: Disk

Base Name: NONE

	Function name	Description
FFE2		[not documented - used by WorkBench 'info']

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

mathffp.library: Fast Floating Point (FFP) arithmetic functions

Name: "mathffp.library"
Version: 33
Revision: 7
IdString: "mathffp 33.7 (6 May 1986)"
NegSize: $0060
PosSize: $0022

Where: KickStart "ROM"

Base Name: MathBase

Function name	Description
FFA0 SPCeil	obtain smallest integer not less than FFP number
FFA6 SPFloor	obtain largest integer not greater than FFP number
FFAC SPDiv	divide two FFP numbers
FFB2 SPMul	multiply two FFP numbers
FFB8 SPSub	subtract two FFP numbers
FFBE SPAdd	add two FFP numbers
FFC4 SPNeg	negate an FFP number
FFCA SPAbs	obtain absolute value of an FFP number
FFD0 SPTst	compares an FFP number with 0.0
FFD6 SPCmp	compares two FFP numbers
FFDC SPFlt	convert an integer to FFP format
FFE2 SPFix	convert FFP number to integer

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

mathieeedoubbas.library: Arithmetic functions using IEEE double precision format

Name: "mathieeedoubbas.library"
Version: 33
Revision: 10
IdString: NONE
NegSize: $0060
PosSize: $0022

Where: Disk

Base Name: MathIeeeDoubBasBase

Function name	Description
FFA0 IEEEDPCeil	obtain smallest integer not less than FP argument
FFA6 IEEEDPFloor	obtain largest integer not greater than FP arg
FFAC IEEEDPDiv	divide two IEEE double precision FP numbers
FFB2 IEEEDPMul	multiply two IEEE double precision FP numbers
FFB8 IEEEDPSub	subtract two IEEE double precision FP numbers
FFBE IEEEDPAdd	add two IEEE double precision FP numbers
FFC4 IEEEDPNeg	negate the supplied IEEE double precision FP num
FFCA IEEEDPAbs	obtain absolute value of IEEE double precis num
FFD0 IEEEDPTst	compares an IEEE DP FP number against 0.0
FFD6 IEEEDPCmp	compares two IEEE double precision FP numbers
FFDC IEEEDPFlt	convert integer to IEEE double precision format
FFE2 IEEEDPFix	obtain integer part of IEEE double precision num

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

mathtrans.library: Fast Floating Point (FFP) Transcendental Functions

Name: "mathtrans.library"
Version: 33
Revision: 8
IdString: NONE
NegSize: $007E
PosSize: $0022

Where: Disk

Base Name: MathTransBase

Function name	Description
FF82 SPLog10	obtain logarithm base ten of FFP number
FF88 SPAcos	obtain arccosine of an FFP number
FF8E SPAsin	obtain arcsine of an FFP number
FF94 SPFieee	convert an IEEE standard FP number to FFP format
FF9A SPTieee	convert an FFP format number to IEEE format
FFA0 SPSqrt	obtain the square root of an FFP number
FFA6 SPPow	obtain the exponentiation of two FFP numbers
FFAC SPLog	obtain the natural logarithm of an FFP number
FFB2 SPExp	obtain e to the power of the FFP number
FFB8 SPTanh	obtain the hyperbolic tangent of the FFP number
FFBE SPCosh	obtain the hyperbolic cosine of the FFP number
FFC4 SPSinh	obtain the hyperbolic sine of the FFP number
FFCA SPSincos	obtain the sine & cosine of the FFP number
FFD0 SPTan	obtain the tangent of the FFP number
FFD6 SPCos	obtain the cosine of the FFP number
FFDC SPSin	obtain the sine of the FFP number
FFE2 SPAtan	obtain the arctangent of the FFP number

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

ramlib.library: RAM-disk handling functions (not called directly by user)

Name:	"ramlib.library"
Version:	33
Revision:	90
IdString:	"ramlib 33.90 (7 Jul 1986)"
NegSize:	$0042
PosSize:	$00B2

Where: KickStart "ROM"

Base Name: NONE

Function name	Description
FFBE	[not documented - internal use only]
FFC4	[not documented - internal use only]
FFCA	[not documented - internal use only]
FFD0	[not documented - internal use only]
FFD6	[not documented - internal use only]
FFDC	[not documented - internal use only]
FFE2	[not documented - internal use only]

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

translator.library: Function to translate text into a phoneme stream

Name:	"translator.library"
Version:	33
Revision:	2
IdString:	NONE
NegSize:	$001E
PosSize:	$002A

Where: Disk

Base Name: TranslatorBase

Function name	Description
FFE2 Translate	converts an English string into phonemes

System summary: Release 1.2 (Kickstart 33.166/Workbench 33.43 - 01-Oct-86)

version.library: This library has a version number, and little else.

Name:	"version.library"
Version:	33
Revision:	43
IdString:	"mylib 33.1 (25 Apr 86)"
NegSize:	$0018
PosSize:	$002C

Where: Disk

Base Name: NONE

There are no library-specific functions in this library.

THE 'KICKSTART' GUIDE TO THE AMIGA™

When Commodore sent out the first issue of "Kickstart - the European Technical Journal" it was hailed as the first thing to explain the machine in a way which was comprehensible to a human being, instead of just to another Amiga!

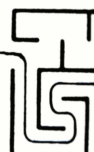
The Commodore Amiga is probably the most advanced wide-market Microcomputer ever produced, both in terms of hardware, and in terms of the system software. The Amiga uses a state-of-the-art message-passing multi-tasking Operating Environment - while this is responsible for a lot of the machine's power, it is also a rich source of confusion to programmers used to comparitively primitive micros.

Aware of this, Commodore commissioned Ariadne to produce the "Kickstart" journal, which was distributed to all European developers. Particularly well received were a series of feature articles, which explained the key concepts of the machine in a way which didn't assume you knew about them already, and which were designed to complement the official documentation as much as possible.

Now with the release of the A500 another group of programmers are eagerly approaching the Amiga. Ariadne have therefore taken the feature articles from Kickstart, revised and updated them, and added new material appropriate to a wider audience - the result is this book.

ARIADNE SOFTWARE LTD

273 Kensal Road, London W10 5DB
Tel: 01-960 0203



This was brought to you

from the archives of

http://retro-commodore.eu