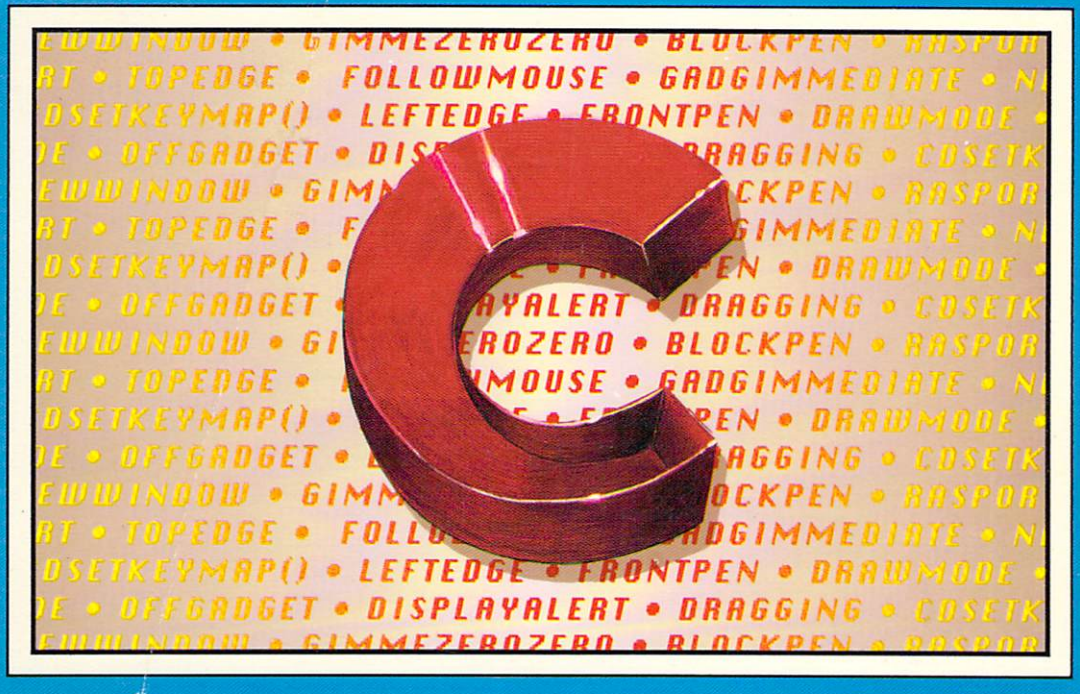
Amiga® C for Advanced Programmers

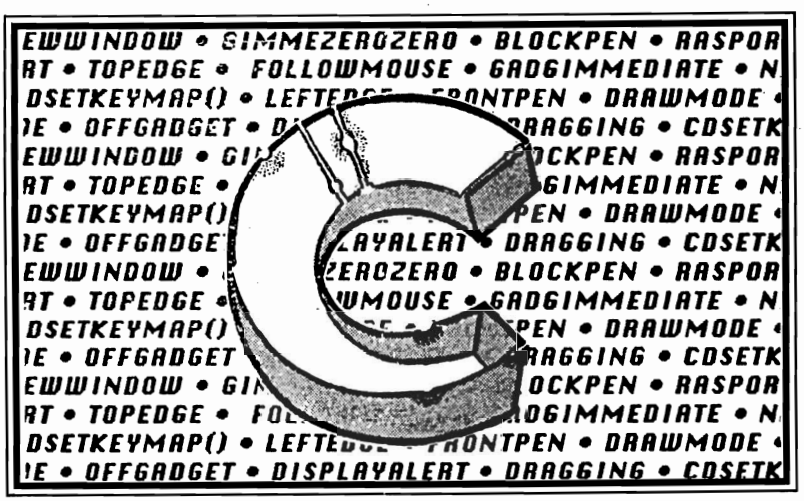
An advanced guide to programming the Amiga using the C language



Abacus
A Data Becker Book

Amiga C for Advanced Programmers

Bleek
Jennrich
Schulz



A Data Becker Book
Published by

Abacus

Second Printing, December 1989
Printed in U.S.A.

Abacus
5370 52nd Street, SE
Grand Rapids, MI 49512

Data Becker GmbH
Merowingerstrasse 30
4000 Duesseldorf, West Germany

Every effort has been made to ensure complete and accurate information concerning the material presented in this book. However, Abacus can neither guarantee nor be held legally responsible for any mistakes in printing or faulty instructions contained in this book. The authors always appreciate receiving notice of any errors or misprints.

ISBN 1-55755-046-8

Table of Contents

Preface

Now that the initial excitement over the Amiga has died down, people are discovering that Amiga can also be used for "real" applications as well as fantastic graphics and sound. Users want more and more literature about programming the Amiga for these serious applications. This book shows you how to make use of the many Amiga operating system functions in your own programs.

This book is not a guide on learning C: It's a book on advanced C programming. If you need a guide for the beginner, please read Abacus' ***Amiga C for Beginners*** before continuing with this book.

Since this book shows you how to program using the C language, we'll begin with author B. Jennrich demonstrating the actual operation of a C compiler. Then we'll get to the actual "instructional" part of the book. You'll learn what makes type casting and other aspects of the language necessary. We hope this saves you time and aggravation in developing larger projects. Maybe you tried C before and spent more time creating errors and system crashes than working programs. This book will hopefully give you a better understanding of the language and what you can accomplish in this language.

The next segment of the book describes the subject of Intuition in detail. W. Bleek, a specialist in this area, leads you into the depths of the Intuition user interface. After reading this section you'll be able to easily add screens, windows, gadgets, menus and more to your C programs.

The third section shows you how to develop an Intuition-based text editor. Peter Schulz, known for his ***AssemPro*** assembler software and the book ***Amiga 3D Graphic Programming*** applies your newly-acquired knowledge of Intuition to a practical project. You'll learn how to work with devices and other libraries.

All in all, this book is one which the serious C programmer will always want to have within reach of his Amiga.

Bleek, Jennrich, Schulz
February 1988

1. The C language

1. The C language

In 1972 Dennis Ritche began work on improving the B language, an improved version of the typeless BCPL language (Basic Combined Programming Language.) The result was the C language.

The C language was developed to ease the implementation of the UNIX operating system on PDP series computers. Dennis Ritche probably never dreamed that his language would become so popular among personal computer users.

C's popularity stems from its nature—an extremely efficient high-level language. This efficiency is especially important in home computing. Home computers contain limited amounts of memory, may require large amounts of time for program execution and have low clock frequencies, therefore a compiler should make a program as small and as fast as possible.

Code created by a C compiler executes quickly and complies into a compact form. These two factors are very important to the programmer who wants to write fast-running programs without the hassle of learning machine language.

In the early days any C source code could be run on any C compiler. This was another factor favoring C—portability. As time went on, however, different operating systems required different compilers (and different operating system interfaces). This made complete compatibility almost impossible.

The 68000-based Amiga, Atart ST and MAC contain many crucial system differences. For example, any Atari ST system routines (GEM, VDI and AES) must be replaced with the equivalent Amiga routines if you want an ST source code to run on the Amiga. This requires a detailed knowledge of both operating systems.

This incompatibility problem becomes worse when you transfer a C source code from an IBM PC or compatible to the Amiga. IBM systems use segmented addressing, requiring the use of FAR declarations of variables and functions. In addition, MS-DOS access occurs almost exclusively through interrupts. This means that the C programmer needs detailed knowledge of both operating systems when converting MS-DOS programs to the Amiga.

1.1 Two Amiga C compilers

Let's leave the compatibility problem for now and turn to the Amiga. You have a choice of basically two complete C compilers for the Amiga—Lattice C and Aztec (Manx) C. Dyed-in-the-wool Lattice programmers swear to the reliability of the Lattice compiler (some parts of the Amiga operating system were written with the Lattice compiler). Meanwhile, Aztec programmers love the speed of their compiler.

The compilers

The earlier versions of the Lattice compiler were considerably better than the early versions of Aztec C. The Lattice compiler could handle arrays which occupied more than 65,535 bytes, while Aztec could not. In addition, problems arose when trying to compile programs written for Lattice on the Aztec compiler. On the other hand, Aztec programs always compiled fine with Lattice. Thus programmers had to stick with the Lattice compiler if they wanted to be able to use their old programs. The new Aztec version (Version 3.4a) has no problems with Lattice compatibility, at least that we could find. We tested a number of programs developed with Lattice and they all compiled successfully with Aztec (with the help of a compiler option).

Thus we decided to use the new Aztec compiler exclusively in this book. The programs will require only minor, if any, modifications to compile with the Lattice C compiler.

1.1.1 The Aztec compiler

Now we come to the Aztec compiler package. The developer's version of the package contains four disks. In addition to the compiler, assembler and linker, these disks contain all of the include files and various libraries which can be linked to compiled source codes and assembler programs (see Section 2.3). These disks also feature several useful utility programs which, for example, allow you to turn your own routines into libraries.

Let's look at what you need for using the compiler. Later versions of the compiler may be different than the descriptions presented here, which are for Version 3.6A. Compilers are continually being updated and improved upon. On disk "`SYS1:`" of the package you will find three subdirectories in addition to a complete Workbench disk. These are accessed using the CLI:

```
bin/
include/
lib/
```

The SYS1:bin/ directory contains all the programs necessary for using the compiler. These include the compiler itself (cc) which creates assembler source. The assembler (as) which assembles this source and the linker (ln) which links the object code of the assembled source with the libraries found in the SYS1:lib directory.

The SYS1:include directory contains the individual header or include files which must be included in each C program. These include and header files access necessary parts of the operating system.

The second disk, "SYS2:" contains assembler include files and additional libraries which you can link with your programs, as necessary (see Section 2.3 for more information). This disk also contains some example programs.

You can start programming in C with only these two disks—the standard version of the Aztec compiler. But the developer's version of the compiler, which adds a few utility programs, makes compilation much easier.

The third disk "SYS3:" contains various utility programs (e.g., make, and more program examples).

Disk "Library Source:", contains the source code of the assembler and C calls for the individual operating system routines in compressed (ARCed) format. These assembler and C files are assembled, compiled and combined into c.lib and m.lib, which can and/or must be linked to your programs.

First we'll look at the standard (and most cost-effective) version of the Aztec compiler. This contains just the disks "SYS1:" and "SYS2:".

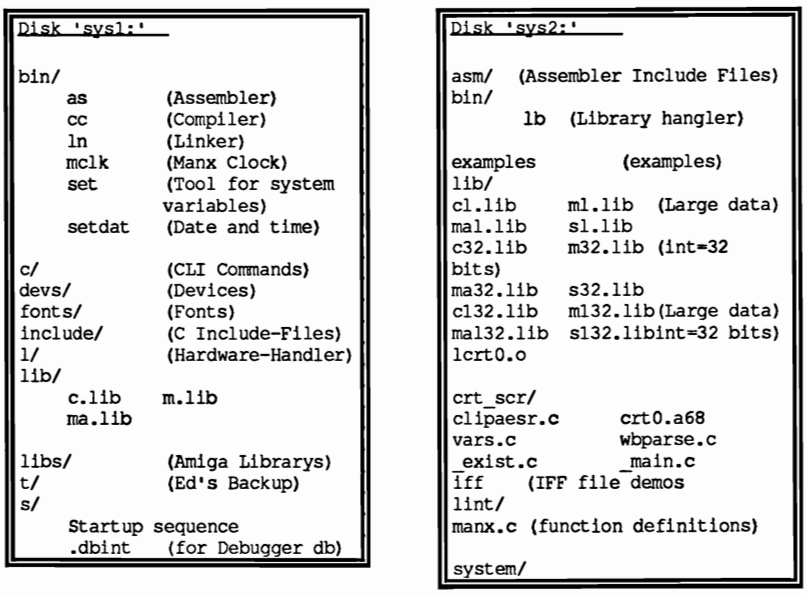


Figure 1.1

Disk 1 is enough to compile short programs. You may have to write a startup sequence to inform the compiler where to find the correct files:

```
;1.1.A.startup-one
stack 10240                     ;stack should be at least 10K
                                ;or more
copy df0:lib/c.lib to ram:
                                ;most common library
                                ;in RAM disk
bin/set  INCLUDE=df0:include
                                ;where are the include files?
bin/set  CLIB=ram:!df0:lib
                                ;where are the libraries?
bin/set  CCTEMP=ram:
                                ;where to put the temporary files?
bin/setdat                      ;set time and date
                                ;(important for developer's!)
run bin/mclk                    ;start time and memory display
```

stack overflow This startup-sequence first increases the stack to 10240 bytes. This is a preventative measure which should protect you from a "stack overflow" when developing programs (you will learn another way of preventing stack overflow later).

linker library Then the startup-sequence copies the linker library (which must be linked to every C program) to the RAM disk. This speeds up the access to the linker library during linking. This library isn't the same as the graphics.library, for example, which contains the addresses of each graphic function in the operating system. The libraries which the linker binds to the C program consist of small machine language segments which call the operating system routines when the system library is used.

But back to the startup-sequence. Next the environment variables of the compiler are set. `bin/set INCLUDE=df0:include` tells the compiler to look in the `df0:include` directory for include files specified in your C program.

CLIB variable The `CLIB` variable performs a similar function. This variable specifies which directory will be searched to find the libraries for linking. `bin/set CLIB = ram:!df0:lib` tells the linker to look for the libraries on the RAM disk first. If it cannot find the files there, the `df0:lib` directory is searched next. If multiple directories are to be searched, the individual directories are separated by "!".

The environment variable `CCTEMP` specifies where the temporary assembly language files are stored. The compiler creates temporary assembly language source files in the specified directory which are later combined into a complete source file and assembled by the assembler.

Next the startup-sequence asks for the current date and time. Those with battery-backed realtime clocks should set the time and date in the startup sequence in the manner appropriate for your clock.

Those who don't own a realtime clock should always enter the correct time and date. You only need to do this when you turn on the computer or first start the compiler. After a reset you can bypass the time and date prompts by pressing <Return>, because the time and date remain somewhat intact on a reset. This only puts the clock behind by a few seconds.

Finally, the `mclk` program is started. This program displays the current time, the free chip and fast memory in a small window in the CLI window's menu line. This is used to check whether the program you are developing actually releases all the memory it allocates. You can get more information about allocated and free memory from the utility program `avail` (`sys3:bin`).

After initialization it's possible to compile small programs with just one disk drive. But you will need two disk drives to compile larger programs. Two disk drives allow you to copy the include files and the linker libraries onto a second disk, for example. You then have to adapt the startup sequence accordingly:

```
;1.1.B.startup-two
stack 10240                    ;stack should be at least 10K
                               ;or more
copy df1:lib/c.lib to ram:     ;most common library
                               ;in RAM disk
bin/set  INCLUDE=df1:include
                               ;where are the include files?
bin/set  CLIB=ram:!df1:lib     ;where are the libraries?
bin/set  CCTEMP=ram:           ;where to put the temporary files?
bin/setdat                     ;set time and date
                               ;(important for developer's version!)
run bin/mclk                   ;start time and memory display
```

Now you have enough room on the disk in drive df0:, disk 1, to write large programs (approx. 3000-4000 lines).

Another possibility would be to leave the startup sequence intact and store the programs on the disk in df1:. This depends on the programmer's preferences. Amiga owners using a hard disk will have to alter their startup sequence accordingly.

1.1.2 The professional version

The professional version of the compiler allows you to create a script file that can be executed with `execute`. This allows compiling, assembling and linking with one line of entry

A very simple script file, which could be named `comp`, looks like this:

```
.key file
cc <file>
ln <file>.o -lRAM:c
```

The source file is first passed to the compiler, which automatically calls the assembler. The assembler then assembles the assembly language source created by the compiler. The object file created by the assembler is then linked with `c.lib` to form an executable program.

This script file assumes that the compiler (`cc`), the assembler (`as`) and the linker (`ln`) are contained in the CLI command directory C: (commands), and can be treated like normal CLI commands. To do this, copy these files from the `SYS1:bin` directory to `SYS1:c`. The `path` command may also be used to set the search path, `path SYS1:BIN`. The program to be compiled must be in the current directory.

Assuming the name of the script file is `comp`, then a C program can be compiled with `execute comp c_prog`. You can interrupt the compiler and linker by pressing <Ctrl><C>. If you call the assembler separately, you can also interrupt script file processing when assembling.

This is because the `FAILAT` level defaults to 10 after loading the Workbench. The compiler and linker interrupt the compilation or link process after <Ctrl><C> with `exit(10)`. This stops the script file processing because it determined through the `exit()` command that a serious "error" occurred. But since the compiler normally calls the assembler, the command `exit(10)` goes to the compiler when <Ctrl><C> is pressed, so the script file cannot be interrupted when assembling.

Another method for interrupting a script file is pressing <Ctrl><D>. This key combination interrupts the script file without stopping any of the programs started from the script file. If the compiler starts and you press <Ctrl><D>, you must wait until the compiler has finished the processing.

1.1.3 The developer's version

Script files are impractical for developing larger programming projects, which normally consist of several modules. For example, if you write a script file which compiles and links all the modules of a program, you have to wait until all the modules have been compiled and linked, even if you made a change in only one of the modules.

You can compile the modified module by hand through a direct call to the compiler, then link it to the other modules "by hand." But this gets rather annoying. This is why the developer's version of the Aztec C compiler contains the `make` tool in addition to the sample programs and libraries.

```
Disk 'sys3:'

lib/
m8.lib      mx.lib          (FFP functions for 6881 and
                            FFP for IEE/Manx
m832.lib    mx32.lib        (Int=32 bit)
m81.lib     mx1.lib         (Large Data)
m8132.lib   mx132.lib       (Large Data and Int=32 bit)

bin/                        (tools)
            ...             ...
            make            db
            ...             ...

examples/   (C examples)
```

Figure 1.2

A `makefile` lets you easily compile large programs consisting of multiple modules. You do not have to manually compile a modified module and then link it with the old modules. `make` handles all this.

You should copy the `make` utility into the C: subdirectory of the "`SYS1:`" disk. Be sure you are working with copies and not the original disks!

Let's look at the following four modules and create a `makefile` to compile them. The `makefile` must be in the same directory as the modules to be compiled and it must be named `makefile`.:

```
/***********************************/
/*  mod1.c                         */
/***********************************/

func1()
{
     printf ("Function 1 calls function 3!\n");
     func3();
}


/***********************************/
/*  mod2.c                         */
/***********************************/

func2()
{
    printf ("Function 2 is called by function 3!\n");
}


/***********************************/
/*  mod3.c                         */
/***********************************/

func3()
{
     printf ("Function 3 calls function 2!\n");
     func2();
}


/***********************************/
/*  main.c                         */
/***********************************/

main()
{
     printf ("MAIN calls function 1!\n");
     func1();
}
```

The `makefile` shows how the individual files depend on each other. For example, the first object file depends on the C source file `mod1.c`. This is expressed in the `make` file in the line `mod1.o: mod1.c`, which says that the commands in the line after `mod1.o: mod1.c` are executed if `mod1.c` is older than `mod1.o` or if `mod1.o` does not exist.

In order to determine which file is older than another, the time and date must be set correctly. The startup sequence calls `setdat`. Since the date of the last modification of a file is always stored in the file info block, `make` can always determine how old a file is.

Back to the `makefile`. The first line of your `makefile` looks like this:

```
mod1.o: mod1.c
```

The next line shows what should be done if `mod1.o` is older than `mod1.c`:

```
    cc mod1
```

If `mod1.o` is older than `mod1.c` (i.e., if a change was made to `mod1.c` since the last time it was compiled), then `make` creates the new object file `mod1.o` (`mod1.o` is updated). The command to be executed according to a file-to-file relationship (`mod1.o: mod1.c`) must be indented by at least one character. The `makefile` which creates the corresponding object files from the four C sources looks like this:

```
mod1.o: mod1.c
 cc mod1
mod2.o: mod2.c
 cc mod2
mod3.o: mod3.c
 cc mod3
main.o: main.c
 cc main
```

Now to create the object file for `main.c`, you only need to call `make main.o`. But you can't use the individual object files if they are not linked. Therefore you must add the following lines to the `make` file:

```
"func: mod1.o mod2.o mod3.o main.o
  ln -o func mod1.o mod2.o mod3.o main.o -lRAM:c"
```

The `func` file These two lines indicate that the `func` file, which represents the executable program relies on the four object files. If one of these files is more recent than `func`, the `func` file must be relinked. The `make` program checks each object file, and if it determines that a modification has been made to one of them, it creates the new object file from the modified C source. This keeps the `func` program up to date after a `make`.

Here is the complete `makefile` which creates the executable function from the modules listed before with `make func`:

```
mod1.o: mod1.c
 cc mod1
mod2.o: mod2.c
 cc mod2
mod3.o: mod3.c
 cc mod3
main.o: main.c
 cc main
func: mod1.o mod2.o mod3.o main.o
 ln -o func mod1.o mod2.o mod3.o main.o -lRAM:c
```

These are `make`'s simplest functions. If you modify the `makefile` so that the linker call (the last two lines) are the first two lines, you can

perform the `make` with just `make` instead of `make func`. When the call is simply `make`, the first command of the `makefile` executes.

If you want to execute multiple commands after a file-to-file relationship, you must ensure that each command is on a separate line and is indented by at least one character. In addition, you must pay attention to the correct calling sequence of commands, whereby you can use any CLI command (e.g., `cd`, `delete`, etc.):

```
func: mod1.o mod2.o mod3.o main.o
 ln -o func mod1.o mod2.o mod3.o main.o -lRAM:c
 delete mod1.o
 delete mod2.o
 ...
```

Another option of `make` is the macro definition. In your `makefile` the string `mod1.o mod2.o mod3.0 main.o` occurs twice. If you define this at the start of the `makefile` you only have to write the string once. This is particularly useful for programs with many modules:

```
OBJECT = mod1.o mod2.o mod3.o main.o

func: $(OBJECT)
 ln -o func $(OBJECT) -lRAM:c
 ...
...
...
```

You don't have to enter the same string repeatedly. If the string doesn't fit on one line, you can "extend" the line with the backslash (\):

```
OBJECT = mod1.o mod2.o mod3.o mod4.o mod5.o mod6.o\
  mod7.o mod8.o mod9.0 mod10.o main.o
```

Now you can start writing your own C programs using the `make` utility.

1.1.4 The commercial version

For the sake of completeness we will briefly describe the commercial version of the Aztec compiler. The only difference between this and the developer's version is that a fourth disk contains the assembly language source for all of the functions. Some support functions for operating devices are written as C procedures.

Most of the machine language calls appear in a form similar to the following:

```
movem.l 4(sp), d0              ;get parameter from stack
...                            ;return address!
move.l #[BasePointer],a6       ;library base in register
jsr _LVOFunction(a6)           ;call function
rts
```

There are also routines which perform more complex computations and manipulations with arguments.

A little tip for those of you who are thinking of purchasing the Aztec compiler: Consider exactly the extent to which you want to program the operating system, and whether it is necessary to buy the commercial version for the function sources. Normally the developer's version will do everything you want, and the large price differential is something to seriously consider.

1.2 The compiler in action

Before we describe the operation of the compiler, we should use it a little and comment on its features.

1.2.1 Comparing files

As an introduction to Aztec programming, we will use a program which compares two files with each other and displays the differences as well as the positions at which they occur. To do this, the program must be told which files to compare. This is done through the command line of the CLI. The arguments of the command line pass to the program as the arguments of the `main()` function:

```
main (argc, argv)
int   argc;
char      **argv;
{...}
```

`argc` (argument count) contains the number of arguments. It should be noted that the program name is interpreted as the first argument. If `argc` equals 2, then actually only one argument was specified.

You can access the arguments (strings) with the pointer `argv`. Let's take a look at this pointer:

```
char **argv;
```

This structure represents a pointer to a pointer of type `char`. A pointer to type `char` is nothing more than a pointer to a string. Because of this close relationship between pointers and arrays, you can access the individual parameters as `argv[1]`, `argv[2]`, etc. `argv[1]` is the address of the first argument string. `argv[0]` contains the address of the program name string. (You can replace `char **argv;` with `char *argv[];`, which verifies array access to the argument strings).

Now let's test to see if the correct number of arguments have been passed. In the example you need two arguments (the names of the two files to be compared). `argc` must have the value 3 (remember the program name!). If it doesn't, a message goes to the user telling him which arguments must be passed.

```
if (argc != 3)
{
   printf ("USAGE: compare FILE1 FILE2 \n");
   exit (10);
}
```

It would be a better programming style to react to a "?" as an argument, printing out some instructions and entering the required parameters direct from the keyboard. Many CLI commands support this option, but it's really not worth the effort to add this to our little example program.

The argument strings are available to you in `argv[1]` and `argv[2]` (`argv` = argument values). You can use these strings as arguments to the C library function `fopen()`.

`fopen()`

`fopen()` requires a second argument in addition to the filename to be opened. This argument specifies how the file is accessed. The function `fopen (name, "r")` opens an existing file for reading, while the function `fopen (name, "w")` creates a new file which can only be written.

In the example you only need be concerned with existing files:

```
File1 = fopen (argv[1],"r");
File2 = fopen (argv[2],"r");
```

Now let's try to open the two files. But since you can't be sure that the user hasn't made a mistake in specifying the files, or that the files even exist, you should test to see if the files actually exist:

```
if (File1 == 0L)
{
    printf  ("%s cannot be opened !!!\n",argv[1]);
    CloseIt (10);
}

if (File2 == 0L)
{
    printf ("%s cannot be opened !!!\n",argv[2]);
    CloseIt (10);
}
```

You may wonder why the letter L follows the 0 in the condition `File1 == 0L`. This tells the compiler that it should compare the pointer `File1` with a long value. You could also replace this `0L` with `(long) 0`.

There is another way to ensure that constants are treated as "long variables": the `+L` compiler option. Here you should note that both constants and `int` variables convert to `long`. When using the `+L` option you should link the `c32.lib` to the program, which ensures that the argument counter (`argc`) of `main(argc, argv)` is actually a long variable.

If you access this counter using `c.lib` and the `+L` option (e.g., `if (argc == 3)`), then four bytes specify the value of this variable instead of just two. However, this completely invalidates the result because the two bytes of the previous `int` variable now become the two high bytes of the current long variable (`+L` option). In addition, the pointers to the strings shift by two bytes.

But back to the real problem: If the file descriptor returned by `fopen()` equals zero, then an error occurred in opening the file. The `CloseIt()` routine closes the open files, sends a message to the user and exits the program. A tip: Programs which allocate memory or perform other operations which must be undone should have a routine which automatically releases everything previously allocated.

If the files were opened properly, you can now compare them. Now comes the main loop of the program, which compares the two files byte for byte for similarities or differences.

The next byte is read from each of the files, and these two bytes are then compared. If the bytes are identical, then the program reads the next two bytes. If the bytes are different, the two bytes and the position in the file at which they occurred are displayed.

It is best to go through the two files inside a `while` loop, which terminates when one of the two files has been completely read. The `feof()` function determines the end of the file, and the next byte of the file can be read with `fgetc`.

```
filepos = 0;
while (!feof(File1) && !feof(File2))
{
  Byte1 = fgetc(File1);
  Byte2 = fgetc(File2);
  if (Byte1 != Byte2)
   printf ("$%08x  %02x <> %02x\n", filepos,Byte1,Byte2);
   filepos++;
}
```

In this loop, the two file bytes (`Byte1` and `Byte2`) are compared with each other and if they differ, a message like the following appears on the screen:

```
$00000154  $12 <> $3a
```

In addition to the two different bytes, their location in the file also appears. This necessitates a counter variable (`filepos`) which contains the number of bytes read, or the read position within the file.

After the comparison, this counter variable increments by one. If both files still contain data, the comparison continues. But if one of the files is empty, the loop ends.

The file attribute which indicates how many bytes the individual files contain comes from the following program segments:

```
if (feof(File1) && !feof(File2))  /* file1 is empty */
                                  /* but File2 is not. */
{
 printf ("Comparison terminated !!!\n");
 printf ("\"%s\" out of data at $%08lx\n",argv[1],filepos-1);
   while (!feof(File2))
   {
      fgetc (File2);
      filepos++;
   }
                                 /* read to end of file */
 printf ("\"%s\" out of data at $%08lx\n",argv[2],filepos-1);
}
else
if (!feof(File1) && feof(File2))  /* File2 is empty */
                                  /* but File1 is not. */
{
  printf ("Comparison terminated !!!\n");
  printf ("\"%s\" out of data at $%08lx\n",argv[2],filepos-
1);
   while (!feof(File1))
   {
      fgetc (File1);
      filepos++;
   }
                                 /* read to end of file */
 printf ("\"%s\" out of data at $%08lx\n",argv[1],filepos-1);
}
```

The two `if` instructions test which file is empty. The name of the empty file and its length are then displayed. The other file is read to the end and for each byte read the `filepos` counter increments. After the end of the file has been reached, this filename and the length of the file are displayed:

```
Comparison terminated !!!
File1 out of data at $00000170
File2 out of data at $00000200
```

A few words about the `filepos` counter. In the first `while` loop, in which the comparison between the two file bytes is performed, `filepos` always increments by one when one of the two files still has bytes available. But if it detects the end of a file with `feof()`, then the `fgetc` inside the loop always reads one more byte than is actually present. This is why when comparing files of different lengths, one extra byte is "compared" and printed. You must ensure that this "extra byte" is subtracted from the size of `filepos` printed.

If both files are the same length, this is determined by the following `if` statement.

```
   else
   if (feof(File1) && feof(File2))
      printf ("\"%s\" and \"%s\" have the size: %08lx\n",
              argv[1],argv[2],filepos-1);
```

The following output then results:

```
FILE1 and FILE2 have the same size: $00000123
```

Now you have to close the open files and exit the program. The `CloseIt()` routine performs this task:

```
CloseIt (Error_Code)
int      Error_Code;
{
   if (File1 != 0) fclose (File1);
   if (File2 != 0) fclose (File2);
   exit (Error_Code);
}
```

This routine closes the open files (file descriptor != 0) then terminates the program with `exit()`. The `Error_Code` which `exit()` passes to the CLI, and can be used to terminate script files.

Here is the complete program, which also defines all of the variables used:

```
/*******************************************************/
/* 1.2.1.A. compare                                    */
/*                        COMPARE                      */
/*                  (c) Bruno Jennrich                 */
/*                                                     */
/*                                                     */
/* This program compares two files with each other.    */
/*******************************************************/
#include "stdio.h"

FILE *File1,                        /* File descriptors */
     *File2;

long filepos;                   /* current file position */

unsigned char Byte1,                          /* byte read */
              Byte2;

main (argc, argv)
int   argc;
char      **argv;
{
   if (argc != 3)
   {
      printf ("USAGE: compare FILE1 FILE2 \n");
      exit (10);
   }

   File1 = fopen (argv[1],"r");             /* open files */
   File2 = fopen (argv[2],"r");

   if (File1 == 0L)                  /* file open() error */
```

```
   {
      printf ("%s cannot be opened !!!\n",argv[1]);
      CloseIt (10);
   }

   if (File2 == 0L)                /* file open() error */
   {
      printf ("%s cannot be opened !!!\n",argv[2]);
      CloseIt (10);
   }

   filepos = 0;            /* current file position = 0 */
   while (!feof(File1) && !feof(File2))
   {
      Byte1 = fgetc (File1);               /* read bytes */
      Byte2 = fgetc (File2);
      if (Byte1 != Byte2)          /* output difference */
         printf ("$%08lx  $%02x <> $%02x\n",
filepos,Byte1,Byte2);
      filepos++;              /* increment file position */
   }

   if (feof(File1) && !feof(File2))  /* file1 is empty */
                                     /* but File2 is not. */
   {
    printf ("Comparison terminated !!!\n");
    printf ("\"%s\" out of data at $%08lx\n",argv[1],filepos-1);
      while (!feof(File2))
      {
         fgetc (File2);
         filepos++;
      }
                                  /* read to end of file */
     printf ("\"%s\" out of data at $%08lx\n",argv[2],filepos-1);

   }
   else
   if (!feof(File1) && feof(File2))  /* File2 is empty */
                                     /* but File1 is not. */
   {
     printf ("Comparison terminated !!!\n");
     printf ("\"%s\" out of data at $%08lx\n",argv[2],filepos-1);
      while (!feof(File1))
      {
         fgetc (File1);
         filepos++;
      }
                                  /* read to end of file */
     printf ("\"%s\" out of data at $%08lx\n",argv[1],filepos-
1);
   }
   else
   if (feof(File1) && feof(File2))
      printf ("\"%s\" and \"%s\" have the size: %08lx\n",
              argv[1],argv[2],filepos-1);

   CloseIt (0);
}

CloseIt (Error_Code)
int      Error_Code;
{
```

```
   if (File1 != 0) fclose (File1);
   if (File2 != 0) fclose (File2);
   exit (Error_Code);
}
```

Since this program consists of just one module, it doesn't make much sense to create a `makefile` for it as well. A script file works just as well:

```
.key file
cc <file>
ln +Cb <file> -Lram:c
```

This script file can (and should) be used for all of the programs in Chapters 1 and 2, if the libraries are in the RAM disk.

The programs created by this script file are usually loaded into the computer's chip memory (`+Cb`). Even if you have more than 512K of memory, the program usually goes into the lower 512K if that memory is available.

If you write a program which doesn't fit in the remaining chip memory (about 320K on the Amiga 1000), you can link your program without the `+Cb` option. However, you must ensure that the structures which lie below the 512K boundary are stored there (such as memory assignments for structures with `*pointer_to_struct = AllocMem(sizeof (struct ...),MEMF_CHIP)`).

1.2.2 File hex dumps

The program above can be easily rewritten so that it outputs a hex dump of a file. A hex dump is program output which appears in the following form:

```
$00000000    $61 $62 $63 $64 $65 $65 $66 $67 $68   abcdefghi
$00000009    $69                                   j
```

The display is the current read position (first column), hexadecimal notation of the characters read (second column), and the characters read in ASCII form.

The program must first be modified so that only one file opens. Here again you must test for the correct number of parameters:

```
main (argc, argv)
int   argc;
char       *argv[];
{
   if (argc != 2)
   {
```

```
    printf ("USAGE: dump FILE \n");
    exit (10);
  }

  _File = fopen (argv[1],"r");

  if (_File == 0)
  {
    printf ("\"%s\" cannot be opened !!!\n",
             argv[1]);
    CloseIt (10);
  }
  ...
}
```

This should be familiar from the last program. What is new is the addition of a conversion table:

```
for (i=0;i<32 ;i++) Conversion_Table[i] = '.';
for (   ;i<128;i++) Conversion_Table[i] = (char)i;
for (   ;i<160;i++) Conversion_Table[i] = '.';
for (   ;i<256;i++) Conversion_Table[i] = (char)i;
```

This conversion table displays the bytes read as characters. You can't display all of the ASCII codes. For example, ASCII code "12" clears the screen. To prevent this sort of thing, all of the printable characters are stored in a table. Unprintable characters (ASCII codes 0-31 and 128-159) are represented by periods "." (see your AmigaBASIC handbook, Appendix A for a list of ASCII characters).

We used four `for` loops to construct this table. Note that the initialization of the loop variable `i` is missing from the last three `for` loops. This is unnecessary. After each `for` loop the loop variable `i` has the same value as the variable used in starting the next `for` loop.

Now the individual bytes can be read, after setting the file position to 0:

```
filepos = 0;
while ((ok = BRead()) != 0)
{
  printf ("$%08lx   ",filepos);
  filepos += ok;

  for (i=0;i<ok;i++) printf ("$%02x ",Byte[i]);

  printf ("  ");

  if (ok < WIDTH) for (   ;i<WIDTH;i++) printf ("    ");

  for (i=0;i<ok;i++)
    printf ("%c",Conversion_Table[Byte[i]]);

  printf ("\n");
}
```

The condition in the `while` loop ensures that bytes are read from the file. To do so, the routine `BRead` is called, which causes either `WIDTH` = 12 bytes or the number of bytes remaining in the file to be read (plus one, because `feof()` doesn't become true until one character has been read past the end).

The variable `z` contains the number of variables read:

```
int BRead()
{
   int i;
   int z;
      for (i=0;i<WIDTH;i++)
     if (!feof(_File))
     {
        Byte[i] = fgetc(_File);
        z++;
     }

   return(z);
}
```

`WIDTH` is a constant defined at the beginning of the program. It specifies the number of bytes to be displayed per line. The sample output at the start of this section was created using a `WIDTH` of 9. When printing the hex dump to an 80-column CLI window, `WIDTH` can be set to 12. This utilizes the entire width of the window.

When the hex dump is displayed, the file position is printed first (`printf ($%08lx ,filepos);`). Next the file position increments by the number of bytes read (`filepos += ok;`). The result of `BRead()` is the number of bytes actually read (`while ((ok = BRead()) != 0)`.

After the file position (and some spaces) has been printed, each byte must be printed in hexadecimal notation. This is done in a `for` loop (`for (i=0; i<ok; i++) printf("$%02x ",Byte[i]);`). Here the hexadecimal numbers occupy two characters. A leading zero is placed in front of the numbers "0" through "f".

If the number of bytes actually read is less than `WIDTH`, spaces are printed for the missing bytes (`if (ok < WIDTH)` for `(i=ok; i<WIDTH; i++) printf("  ");`). It can only occur at the end of the file that fewer than `WIDTH` characters could be read, but it still looks unprofessional if the hexadecimal and ASCII characters are not kept in their respective columns, such as:

```
$00000000 $12 $23 $10 $07 $06 $07 $c2 $82 $85   .........
$00000009 $12 $23 $10 $07 $06    .........
```

Therefore the `for` loop prints as many spaces as would be used by a hexadecimal number.

Now you just have to print the bytes as ASCII characters (`for (i=0; i<ok; i++) printf("%c",Conversion_Table [Byte[i]]);`). The previously constructed `Conversion_Table` is indexed by the previously read bytes. After the ASCII characters have been printed, a carriage return is printed (`printf("\n");`) so that subsequent output appears on the next line.

The file must be closed when the `while` loop exits (all bytes of the file have been read). The modified version of `CloseIt()` used in this program looks like this:

```
CloseIt (Error_Code)
int      Error_Code;
{
   if (_File == 0) fclose (_File);
   exit (Error_Code);
}
```

Here is the entire program, again with the definitions of all variables and arrays:

```
/*******************************************************/
/*                                                     */
/*  1.2.2.A. dump.c                                    */
/*                          DUMP                       */
/*                  (c) Bruno Jennrich                 */
/*                                                     */
/*                                                     */
/* This program creates a hex dump of a file.          */
/*******************************************************/
#include "stdio.h"

#define WIDTH 12L              /* 12 numbers per line */
                                  /* 80-column screen */

FILE *_File;                       /* file descriptor */
long filepos;                 /* current file position */
unsigned char Byte[WIDTH];    /* buffer for bytes read */
unsigned char Conversion_Table[256];
                      /* ASCII - printable conversion */
int ok;                        /* number of bytes read */

int BRead()
{
   int i;
   int z = 0;

   for (i=0;i<WIDTH;i++)
      if (!feof(_File))
      {
         Byte[i] = fgetc(_File);
         z++;
      }

   return (z);
```

```
}

main (argc, argv)
int   argc;
char       *argv[];
{
   int i;                                /* loop variable */

   if (argc != 2)                  /* too few parameters */
   {
      printf ("USAGE: dump FILE \n");
      exit (10);
   }

   _File = fopen (argv[1],"r");             /* open file */

   if (_File == 0)                    /* file open error */
   {
      printf ("\"%s\" cannot be opened !!!\n",argv[1]);
      CloseIt (10);
   }

   for (i=0;i<32 ;i++) Conversion_Table[i] = '.';
   for (   ;i<128;i++) Conversion_Table[i] = (char)i;
   for (   ;i<160;i++) Conversion_Table[i] = '.';
   for (   ;i<256;i++) Conversion_Table[i] = (char)i;
                           /* Conversion_Table aufbauen */

   filepos=0;         /* current file position equals 0 */
   while ((ok = BRead()) != 0)
   {
      printf ("$%08lx   ",filepos);
                          /* print current file position */
      filepos+=ok;  /* increment current file position */

      for (i=0;i<ok;i++) printf ("$%02x ",Byte[i]);
                                 /* hexadecimal numbers */

  if (ok < WIDTH) for (i=ok;i<WIDTH;i++) printf ("    ");
                                       /* "blank numbers" */
      printf ("  ");                    /* column divider */

      for (i=0;i<ok;i++) printf
         ("%c",Conversion_Table[Byte[i]]);
                               /* "ASCII" representation */
      printf ("\n");                   /* carriage return */
   }
```

```
   CloseIt (0);                                    /* bye */
}

CloseIt (Error_Code)
int      Error_Code;
{
   if (_File != 0) fclose (_File);     /* if file open */
                                          /* close it */
   exit (Error_Code);
}
```

1.2.3 A monitor

With a few changes, the hexadecimal dump program can also be used as a machine language monitor. A monitor lets you view the contents of the Amiga's memory. The monitor displays the various memory contents.

To do this you must tell the program the starting and ending addresses of the area of memory you want to see. This is done again with command arguments. Two arguments are passed:

```
main (argc, argv)
int   argc;
char      **argv;
{
   if (argc != 3)
   {
      printf ("USAGE : mon START END \n");
      exit (10);
   }

...
}
```

After you have determined that the correct number of arguments have been passed, you can start converting them. The starting and ending addresses are specified as ASCII strings, and must be converted to numbers. We decided to convert the strings to hexadecimal numbers. Almost every monitor on the market works with hexadecimal numbers, so your monitor will too.

The Aztec compiler offers a routine which converts an ASCII string into a hex number. The following is our conversion routine:

```
unsigned char *atoh (String)
char               *String;
{
   int i;
   unsigned char *hex;
```

```
    unsigned long factor;

    hex = (unsigned char *)0L;
    factor = 1L;

    if (strlen (String) <= 8)
      for (i=(strlen(String)-1);i>=0;i--)
      {
           if ((String[i] >= '0') && (String[i] <= '9'))
           {
                hex += (String[i] -'0')*factor;
                factor *= 0x10;
           }
                if (((String[i] & MASK) >= 'A') &&
               ((String[i] & MASK) <= 'F`))
           {
             hex += ((String[i] & MASK) - 'A'+0xa)*factor;
                factor *= 0x10;
           }
      }
    return (hex);
}
```

This routine first sets the return value `hex` to 0 and `factor` to 1. The following steps only occur when the string length doesn't exceed eight bytes. This is because the address range of four gigabytes, which the Amiga can theoretically address, is represented by the addresses from 0 to FFFFFFFF. The highest address is therefore FFFFFFFF. As a string, this number is eight characters long.

If the string passed to the routine has eight or fewer characters, the `main` loop of this function executes. In this loop the string is examined character by character—starting with the last character of the string and working toward the first.

In this loop, the value 1 is subtracted from the length of the string in the initialization of the loop variable (`for (i=(strlen(String)-1);i>=0;i--)`). This is done because the first character of the string is `String[0]` and not `String[1]`. So you don't have to subtract one from subsequent accesses to characters within the loop, you do it once and for all at the initialization of the loop variable.

The loop distinguishes between two cases:

1. The character of the string is a number (0-9).

2. The character of the string is a letter (A-F or a-f).

When calculating the result you have to test to see if the current character is a digit (0-9) or a hex character (0-F).

In the first case the ASCII value of the digit 0 is subtracted from the ASCII value of the digit 0. This is done to obtain the numerical value

which the digit represents. "1 - 0" yields the number 1. "2 - 0" yields the number 2, etc. These relationships can be easily seen from an ASCII table. The ASCII code for the digit 0 is 48, the digit 1 is 49, and so on, up to the digit 9, which has the ASCII code 57.

This number is multiplied by `factor` and added to the previous result. `factor` is then multiplied by the value "0x10 = 16". This causes a place shift.

In the decimal system the number 123 means:

```
3 ones + 2 tens + 1 hundred
```

This is similar for hexadecimal numbers, except that the number 16 is used as the base instead of 10. The hexadecimal number "123" means:

```
3 ones + 2 sixteens + 1 two-hundred-fifty-sixes
```

Or written differently:

```
3 * 0x1 + 2 * 0x10 + 1 * 0x100
```

`factor` runs through `0x1`, `0x10`, `0x100`, etc. Now it may be clear why the string processing goes from back to front. This guarantees that the last character of the string represents the ones place of the number. The second-to-last character is in the sixteens place, the third at the 256's place, etc.

Now we come to the second case. Here you have a hex digit A-F. Since you do not assume that you use only upper or lowercase, you first have to convert the character to uppercase. This is done by masking out bit 4. (`String[i] & MASK" = "String[i] & (255L-32L)`) (see ASCII table). The ASCII value of the letter "A" is subtracted from the hex digit in order to get the numerical values 0, 1, 2 etc. Since the digit A corresponds to the value `0xa` or 10, you have to add `0xa` to this result. You can continue the calculation as above.

The starting and ending addresses which determine the memory area can now be calculated with this function:

```
lowadress = atoh (argv[1]);
highadress = atoh (argv[2]);
```

You should note that characters which aren't valid hex digits are ignored. The string 12$$5 returns the same value as 125.

Now test to see if the starting address is lower than the ending address of the memory range. The memory is processed in ascending order:

```
if (lowadress > highadress)
{
   printf ("START must be less than END !\n")
   exit (10);
}
```

If the starting address is larger than the ending address, the program exits immediately. If the starting address is less than the ending address, then you can use the conversion table from the previous program example:

```
for (i=0;i<32 ;i++) Conversion_Table[i] = '.';
for (   ;i<128;i++) Conversion_Table[i] = (char)i;
for (   ;i<160;i++) Conversion_Table[i] = '.';
for (   ;i<256;i++) Conversion_Table[i] = (char)i;
```

Similar to the way you used `filepos` in the previous programs to determine the current file position, use `pos` here to store the value of the memory location to display. At the start this contains the value of `lowadress`:

```
pos = lowadress;
```

Now you can start displaying the memory contents. This should be done in a separate routine because you have to test when the output should stop.

In the hex dump of a file it wasn't so hard because the `Read()` function would tell you if you could continue to read from the file or not. `Read()` also told you how many bytes had to be printed.

You have to calculate the number of characters to display yourself:

```
width = WIDTH;
while (pos < highadress)
   if ((highadress-pos) >= width) Show (WIDTH);
   else Show (highadress-pos);
```

As long as the value of `pos` has not reached the address of the highest memory location to display, bytes are displayed. You also need to check to see if you have enough bytes to fit in a line (`Show(WIDTH)`) or fewer.

If the number of bytes to print is less than the number of bytes which can be displayed on a line (`WIDTH`), then only the remaining bytes are displayed (`Show (highadress-pos)`).

To test to see if fewer than `WIDTH` bytes are remaining to be printed, you must compare the number of bytes to be printed (`highadress-pos`) with the number of bytes which can be displayed on a line. Unfortunately, the Aztec compiler does not understand long constants (like 12L) in a comparison of long constants (an exception is "==" in

which long constants are not processed). Long constants are correctly handled in assignments and as parameters to procedures.

We used a little trick: We assigned the value of the long constant to a long variable and used this variable in the comparison. The compiler understands this (you don't have to use this trick with the +L option and the c32.lib library).

Now on to the routine responsible for printing the bytes on the screen. It should look familiar to you—we made a few small changes:

```
Show (ok)
long  ok;
{
   int i;

   printf ("$%08lx    ",pos);

   for (i=0;i<ok;i++)
      printf("$%02x ",*(pos+i));

   if (ok < WIDTH) for (   ;i<WIDTH;i++) printf ("    ");

   printf (" ");

   for (i=0;i<ok;i++)
     printf ("%c",Conversion_Table[*(pos++)];

   printf ("\n");
}
```

This routine first outputs the address of the first byte to be displayed on the line. Then each byte is printed in hexadecimal notation. The method used to make sure that the ASCII versions of the bytes are always aligned when there are fewer then WIDTH characters to be printed is also familiar.

The access to pos is something new. This variable (a pointer) is accessed with *(pos+i). This means: Add the value i to the current address stored in pos, and return the contents of the memory location to which this result points.

The same goes for *(pos++). This means: Increment the value of pos by one and return the contents of the memory location to which this value points.

The parentheses are necessary in both cases. * has a higher precedence than ++. If the parentheses were omitted, *pos++ would mean: Get the contents of the memory location to which pos points and then increment the value of pos.

Here again is the complete program:

```
/*************************************************************/
/*                      1.2.3.A.mon.c                        */
/*                                                           */
/*                         MON                               */
/*                   (c) Bruno Jennrich                      */
/*                                                           */
/*                                                           */
/* This program outputs a memory range                       */
/* as a hex dump.                                            */
/*************************************************************/

#define WIDTH 12L              /* number of bytes to display */
#define MASK (255L-32L)       /* for conversion to uppercase */

unsigned char Conversion_Table[256];
                                          /* conversion table */
unsigned char *lowadress,   /* lowest address to be displayed */
              *highadress; /* highest address to be displayed */

unsigned char *pos;           /* current address or position */
                                              /* in memory  */

unsigned char *atoh (String)          /* digit string to hex */
char               *String;
{
   int i;
   unsigned char *hex;                              /* result */
   unsigned long factor;                      /* place factor */

   hex = (unsigned char *)0L;                     /* result = 0 */
   factor = 1L;                /* factor equals 1 (one place) */

   if (strlen (String) <= 8)    /* more than eight characters */
      for (i=(strlen(String)-1);i>=0;i--)
      {
         if ((String[i] >= '0') && (String[i] <= '9'))
         {                                             /* figures */
            hex += (String[i] - '0')*factor;
            factor *= 0x10;
         }

         if (((String[i] & MASK) >= 'A') &&
             ((String[i] & MASK) <= 'F'))
         {                     /* Hex-figures */
            hex += ((String[i] & MASK)-'A'+0xa)*factor;
            factor *= 0x10;
         }
      }

   return (hex);
}

main (argc, argv)
int   argc;
char      **argv;
{
   int i;
   if (argc != 3)            /* number of parameters is wrong */
   {
      printf ("USAGE: mon START END\n");
      exit (10);
   }
```

```
  lowadress  = atoh (argv[1]);          /* calculate lowest and */
  highadress = atoh (argv[2]);             /* highest addresses */

   if (lowadress > highadress)                /* error in input */
   {
      printf ("START must be less than END !!!\n");
      exit (10);
   }

   for (i=0;i<32 ;i++) Conversion_Table[i] = '.';
   for (   ;i<128;i++) Conversion_Table[i] = (char)i;
   for (   ;i<160;i++) Conversion_Table[i] = '.';
   for (   ;i<256;i++) Conversion_Table[i] = (char)i;
                                  /* build conversion table */

   pos = lowadress;                /* current position equals */
                                            /*lowest address */

   while (pos < highadress)                     /* output loop */
      if ((long) (highadress-pos) >= WIDTH) Show (WIDTH);
      else  Show ((long) (highadress-pos));
}

Show (ok)
long ok;
{
   int i;

   printf ("$%08lx   ",pos);               /* output position */

   for (i=0;i<ok;i++)
     printf ("$%02x ",*(pos+i));        /* output byte as hex */

   if (ok < WIDTH) for (   ;i<WIDTH;i++)
      printf ("    ");                        /* fill space with */
                                                    /* blanks */
   printf (" ");

   for (i=0;i<ok;i++)
      printf ("%c",Conversion_Table[*(pos++)]);
                                              /* bytes as hex */

   printf("\n");                             /* carriage return */
}
```

Note that the contents of the specified ending address are not printed. `mon 100 200` displays the contents of memory locations `hex 100` to up to and including `hex 1ff`.

2.
C Compiler Operations

2. C Compiler Operations

The compiler, assembler and linker are all common to the two popular C compiler implementations. We will now look at these three and their options and take a closer look at the compiler during compilation.

First let's look at how the compiler is called. It follows this form:

```
cc [>output_file] [options] prog[.c]
```

Everything in square brackets is optional. You must specify the name of the file to be compiled. It doesn't matter to the compiler if you enter `cc program.c` or `cc program`.

The `output_file` and `options` arguments can be optionally specified. The `output_file` is especially useful during program development. All messages displayed by the compiler (including error messages) are written to this file. For example, if you enter `cc >prt: program` the following is sent to the printer, assuming that no errors occur in the program:

```
"Aztec C68K 3.6a  12-18-87 (C) 1982-1987 by Manx Software
                                              Systems, Inc.
Aztec 68000 Assembler 3.6a  12-18-87"
```

You can also send the messages to a disk file named `errors.c` and then read this file with ED (`cc >errors.c program`).

Redirection

The ability to send the output to another file or device is called *redirection*. It can be used with almost any CLI command, including the assembler and linker. For example, `dir >prt:` sends the current directory to the printer.

When redirecting compiler messages, you should bear in mind that when five errors (not warnings) have occurred, the compiler asks the user if he wants to continue compiling. This has the advantage that error messages don't scroll quickly off the screen. You can see all of the error messages, even if there are too many to fit on the screen at a time.

This is unnecessary when sending the messages to the printer or a file. You should disable this feature using the compiler option -B.

An error message always has the following format:

```
Listing_of_the_line_in_which_the_error_occurred
 ^

program_name.c:error_line: ERROR number: description:
```

```
/* example of error message */
}
 ^
test.c:23: ERROR 69: missing semicolon
```

The caret (^) shows where the compiler believes the error occurred. The compiler doesn't notice the missing semicolon error, number 69, until one line later. Since it is an unwritten law that only one function is written per line (excluding `if` statements where commands only appear in the "then" segment of the function), this error always occurs one line later.

Now we come to the compiler options:

Aztec Compiler Options

`-A` The assembler doesn't automatically start when the compiler is finished.

`-Dsymbol[=value]`
This option assigns the optional value to the specified symbol. `-DNAME=value` is the same as `#define NAME value`, while `-DNAME` causes the symbol to represent the value 1 (`#define NAME 1`).

`-Idirectory` The `include` files are read from the specified directory. If you want to search multiple directories, you can separate directories with exclamation points (`-Idir1!dir2!dir3`).

`-O filename` The object file (or the assembler source, if the `-A` option is used) is written to the specified file.

`-S` Warning messages are suppressed (silent option).

`-T` The lines of C source code are included in the assembler source as comments. This option only makes sense with the `-A` option.

`-B` This option suppresses the halting of message output after every five messages. When directing error messages to the printer, for example, it doesn't make sense for the compiler to stop after every five messages to ask you if you want to continue.

`-Enumber` This option allocates an expression table with numeric entries. This expression table is used when the compiler evaluates `for` loops, `switch/case` statements, etc.

`-Lnumber` This option reserves numeric entries for the local symbol table. This local symbol table is used for variable definitions and declarations within func-

tions or blocks (program segments are enclosed in curly braces).

`-Ynumber` This option allocates a case table with numeric entries. This `case` table is used when compiling `switch` statements.

`-Znumber` This option allocates a string table with `number` bytes. The string table is used for storing strings during compilation.

`+B` This option suppresses creation of `public .begin` in the assembly language file. This is important when creating multiple modules to prevent the command `jmp .begin` from being included in every module.

`+C` Creates "large code" (see Section 2.3).

`+D` Creates "large data" (see Section 2.3).

`+Hfilename` This option saves the symbol table created by the compiler for a program.

`+Ifilename` This option loads the symbol table previously saved with `+H`. This allows you to compile, save and reload the `include` files used by a program. This increases the speed of later compilations because the `include` files don't need recompiling each time.

`+L` This option instructs the compiler to interpret all `int` variables and constants as `long`. Together with the `c32.lib` library, this configures Aztec C to compile source codes originally written for Lattice.

`+Q` The strings used in the program are written to the data segment.

`+ff` Calculations are performed in single precision. `m.lib` must be linked with the object file.

`+fi` Calculations are performed in double precision. You must link in `ma.lib` or `mx.lib` when using the double precision floating point libraries.

`+f8` This option enables the 68881 math coprocessor for calculations. When using this coprocessor, which must naturally be installed, you must link in the `m8.lib`.

`+m`	This option checks the stack for overflow. If overflow occurs, a runtime error is produced.
`-n`	Debugging information is added to the assembler source, which is further processed by the assembler.
`+P`	This option is identical to the options `+C +D +L`. In addition, the registers D2 and D3 are saved before each function call. This option requires that the `cl32.lib` be linked with the main program.
`+r`	The A4 register can be used as a register variable. This option requires using the large code and data model and that the `cl.lib` or `cl32.lib` be linked with the main program.

Some of these options are self-explanatory, such as the `-A` option which you can remember as "minus assembler". It tells the compiler not to start the assembler as it usually does.

When the `-A` option is used, you still have to tell the compiler where to write the assembly language source file which it normally creates on the RAM disk (see `CCTEMP` variable) and deletes later. You must specify the name of the assembly source file with the `-O` option. The compiler and assembler can then be called separately:

```
cc -A -O program.asm program.c
as program.asm
```

It's more than just passing additional arguments to `cc`. You also have to call the assembler "by hand," which creates the object file `program.o` from `program.asm`. If you specify only the `-O` option (without -A) when compiling, the object file created by the assembler (now called by the compiler) has the specified name.

Normally you don't have to specify the name for the object file because it automatically has the same root name as the C source file, but with an extension of `.o`. If your program is large enough that you have to conserve every byte of space on the disk, you should remove the object file after each link with `delete`. This can be done with the professional version in the script file or with the developer's version in the `makefile`. This is useful only when you have multiple programs on the disk, each of which consists of only one module.

2.1 How a C compiler works

In order to explain the additional compiler options, lets take a closer look at the compiler itself. We want to see what the compiler does with a source code, which is really only an ASCII file.

First, the compiler looks for obvious errors. The compiler knows, for example, that a semicolon must follow each statement. If the compiler does not find the semicolon after a statement, it displays an error message. If the condition is missing in an `if` statement, the compiler tells the user. However, the compiler cannot recognize all other errors. These are the errors which only you as the programmer can determine. These errors usually manifest themselves in system crashes, or when the compiled program doesn't do what you expected it to do. Then the debugger comes into play (see Section 2.5).

A typical error of this type is exchanging "=" for "==", using "=" in the condition of an `if` statement when "==" is intended. Another common error is accessing a non-existent array element:

```
int array[4];
```

This array contains the elements 0, 1, 2 and 3, for a total of four elements. But if you assign a value to `array[4]`, the "fifth" element, you may change another program variable, or even cause the program to crash.

When hunting down these errors, it helps to have a good eye and a feel for where such errors can hide. Failing good eyes and programming know-how, the debugger can help.

After the error test comes the translation of statements and expressions into assembly language. This translation can take place immediately after checking a statement for errors, or after checking the entire program, depending on the compiler.

2.1.1 Internal organization of variables and structures

The program translation can begin if the program is free of syntax errors, etc. Generally a C program consists of variable definitions and declarations in addition to statements.

Before looking at actual C statements, let's look at how variables and structures are compiled:

First the individual variable types must be determined. The C language recognizes the following scalar types:

```
int, long, char, short, unsigned char, unsigned int,
unsigned long, float, double and pointers
```

Structured types are arrays (or vectors), `structs`, enumerated types, bit fields and `unions`.

The compiler encodes each type and stores the variable name and its type code in the global symbol table. It should be pointed out here that the following symbol tables are for an ideal C compiler and not the Aztec compiler. But the Aztec compiler works according to the same principle as the ideal compiler.

```
int i_val;
char c_val;

Symbol table:
-------------
"i_val"/code_int/end_code/"c_val"/code_char/end_code
```

It looks similar for structures. Here you have the name of the structure, followed by a code (a single byte) for the start of the structure. Then the variables contained in the structure are listed:

```
struct nonsense
{
   int hello;
   char na;
   unsigned char good_day;
};

Symbol table:
-------------
"nonsense"/"code_structure_type/"hello"/code_int/
"na"/code_char/"good_day"/code_unsigned_char/end_code
```

This is a structure declaration. The individual codes are represented here with text (e.g., `code_char` instead of a `byte` value). The structure

cannot be used until a suitable variable has been defined (`struct nonsense nonsense_structure`). This is also possible with `typedef` (`typedef struct {...} structure_type_name`). Scalar types can also be renamed with `typedef`:

```
"typedef unsigned char BYTE"

Symbol table:
-------------
"BYTE"/code_type_begin/code_unsigned/code_char/end_code
```

The `include` file `exec/types.h` contains a number of new types which are used in almost every `include` file. They are required for using `include` files and must be included before any of the other `include` files can be used.

But back to structures. The symbol table for a structure definition looks like this:

```
struct nonsense nonsense_structure;

Symbol table:
-------------
"nonsense_structure"/code_structure_begin/
"nonsense"/code_structure_type/end_code
```

You can directly define a `nonsense_structure`. The structure is then no longer available for future definitions:

```
struct
{
   int hello;
   char na;
   unsigned char good_day;
} nonsense_structure;

Symbol table:
-------------
"nonsense_structure"/"code_structure_begin/
"hello"/code_int/"na"/code_char/
"good_day"/code_unsigned_char/end_code
```

A definition of the form `struct nonsense_structure` name is no longer allowed here. The declaration of bit fields, arrays, enumerated types and `unions` is similar to that of structures:

```
ARRAY:
------
int digits[6];

Symbol table:
-------------
"digits"/code_array/code_int/6/end_code
```

```
ENUMERATED:
------------
enum color = {red, green, blue};

Symbol table:
------------
"color"/code_enum/"red"/0/"green"/1/"blue"/2/end_code
```

Compiler assigned integer numbers represent the individual enumerated types in the program. The same effect can be achieved with a set of `#define` instructions. Instead of `enum color = {red, green, blue}` you could write:

```
#define red 0
#define green 1
#define blue 2
...
int color;
```

It should be noted that a `#defined` symbol simply replaces it with the string which follows it. In the symbol table, the values of `#defined` symbols are therefore stored as strings:

```
Symbol table for defines:
-------------------------
"red"/code_define/"0"/end_code/"green"/code_define/
"1"/end_code/"blue"/code_define/"2"/end_code
```

Caution should be used especially for mathematical expressions:

```
#define sum a+b
...
printf("%d  %d",sum,sum*10);
```

The `printf` function first prints the sum of the two variables `a+b`. If the second output is supposed to be the sum multiplied by ten, you will probably be disappointed here. The compiler sees `sum*10` and generates `a+b*10`. It adds the value of `a` to ten times the value of `b`. You should put the equation in parentheses to avoid this:

```
#define sum (a+b)
...
```

The `#ifdef`, `#ifndef` and `#endif` directives should also be mentioned in the context of `#define`. These occur often in include files and are usually used to ensure that an include file compiles only once, since otherwise this would lead to errors (redefined symbols, etc.). Since most include files are built from others, includes often appear within include files:

```
/* **** Include-File **** */

#ifndef EXEC_TYPES_H          /* if EXEC_TYPES_H is not */
                                        /* defined, */
```

```
#include "exec/types.h"   /* then include exec/types.h */
#endif

#ifndef GFX_H              /* if GFX_H is not defined, */
#include "graphics/gfx.h" /* then include graphics/gfx.h */
#endif
```

The compiler can use the symbol table to check to see if a symbol is defined (`#ifdef`) or not (`#ifndef`). The compiler can then determine the fate of the statements enclosed in the `#if...` – `#endif` block. Each include file defines a symbol at the start with the name "`IncludeFile_H`".

Bit fields

Now let's look at bit fields. These are a new feature added to Aztec compiler version 3.4a. As the name implies, bit fields provide a way of organizing individual bits of information. They are declared similarly to structures:

```
struct bitfield
{
   unsigned 2:;      /* bits 0 and 1 */
   unsigned 1: a;    /* bit 2 */
   unsigned 1: b;    /* bit 3 */
   unsigned 1: c;    /* bit 4 */
   unsigned 3:;      /* bits 5-7 */
};
struct bitfield bf;
```

This declaration creates an eight-bit-wide bit field. You can now access bits 2, 3 and 4 with `bf.a`, `bf.b` and `bf.c`. You can only set these variables to 0 or 1. This is logical, since a single bit can have only one of two values.

You cannot access bits 0, 1, 5, 6 and 7 because they were not assigned names. These bits were "skipped" as told by the `unsigned 2:;` and `unsigned 3:;` directives.

The symbol table for bit field declaration and definition above looks like this:

```
Symbol table:
-------------
"bitfield"/code_type_begin/""/code_bit0/""/code_bit1/
"a"/code_bit2/"b"/code_bit3/"c"/code_bit4/""/code_bit5/
""/code_bit6/""/code_bit7/end_code

Definition:
-----------
"bf"/code_structure_start/code_type/"bitfield"/end_code
```

`union`

Now let's look at the last of the non-scalar data types—the `union` `union` designates a variable which can consist of more than one data type:

```
union number
{
   int i;
   float f;
}
union number n;

Symbol table:
-------------
"number"/code_union_type/"i"/code_int/"f"/code_float/
end_code

Definition:
-----------
"n"/code_union_type/"number"/end_code
```

The variables `i` and `f` occupy the same storage space. Therefore, this variable can be interpreted as either type.

There are also pointers to the individual variables, structures and arrays, so we must introduce the code for pointers:

```
int *pointer;
struct nonsense *sensible;

Symbol table:
-------------
"pointer"/code_pointer/"sensible"/code_int/end_code/
"sensible"/code_pointer/"nonsense"/code_structure_type/
end_code
```

The symbol table constructed during the compilation is needed whenever a variable is accessed. If you want to increment the variable `i` (`i++`), for example, the compiler must know the variable type of `i`.

Assuming that `i` is an `int` object, the compiler then knows that this variable is 16 bits wide and that it must generate the statement `add.w #1,address_of_i` and not `add.b #1,address_of_i` or `add.l #1,address_of_i` for the assembly language source.

It is similar when accessing the values of structure elements, e.g., `sensible->hello = 0`. The compiler must first find out if `sensible` is really a pointer. From the symbol table it learns: `sensible/code_pointer` and knows that `sensible` is a pointer. Now it must determine what kind of object `sensible` points to. If `sensible` points to an `int` variable, than the line `sensible->hello` doesn't make sense, because `int` variables don't contain additional elements, as do structures.

From the symbol table the compiler discovers that `sensible` points to a structure. The "->" is therefore allowed. `sensible->hello` can also be replaced with `(*sensible).hello`, whereby it should be

noted that the period (.) has higher priority than the star (*) in front of `sensible`, which is why the parentheses are necessary.

If you wrote `sensible.hello`, the compiler would respond with an error message (incorrect pointer usage).

Now the compiler must check to see if the structure to which `sensible` points has an element named `hello`. It also gets this information from the symbol table. First `sensible` points to a `nonsense` object, and in the symbol table entry for `nonsense` the element `hello` occurs. Now the compiler just has to check to see if the assignment of 0 is allowed to the variable `hello` of the `nonsense` structure, and if so, it must determine what this assignment should look like. After the compiler has determined from the symbol table entry for `nonsense` that `hello` is an `int` object, it can produce `clr.w address_of_sensible->hello`.

During the compilation, the compiler checks the type of each variable whenever it is accessed. If you try to assign a variable to a variable of a different type, the compiler produces a warning which tells you that an illegal type assignment is being performed.

The compiler automatically performs the appropriate type conversions when assigning standard types like `char`, `int` and `long`. In the assignment of an `int` object to a `char` variable, for example, only the lower eight bits of the `int` object are passed to the `char` variable, while the missing bits in the assignment of `char` to `long` are filled with zeroes (signs are added as needed).

Type conversion becomes problematical in connection with pointers:

```
int *Input1;
char *string = "The Amiga is tops!!";

string = Input1;
```

The compiler isn't certain if the programmer knows that he has assigned two pointers with different types. Therefore, the compiler displays a warning which warns the user about a potentially incorrect pointer assignment.

It can be desirable to assign an `int` pointer to a `char` pointer, perhaps to access the high-order byte of the `int` variable through the `int` pointer. This is possible with a `char` pointer because `char` objects are only one byte wide and after `string++` the pointer `string` points to the next address.

If you want an assignment like this to occur, you should let the compiler know this with the help of a type `cast`:

```
string = (char *) input;
```

The program segment above illustrates two features beside the cast: First, a variable is initialized in its definition, and second, strings are used.

When initializing a variable the location at which the variable definition rests is important. If the variable was defined outside a function, the storage space occupied by this variable is assigned with dc.<b/w/l> value:

```
/**************************/
/* C code                 */
/**************************/

char *string = "The Amiga is tops!!";


/**************************/
/* Assembly language      */
/**************************/

   dseg                              ;data segment
   ds 0                              ;reserve 0 bytes
   public _String                    ;global variable
_String:                             ;allocate memory
   dc.l .1+0                         ;with address of string
   cseg                                      ;code segment
.1
   dc.b 84,104,101,32,65,109,105,103,97,32,105,115,32,84
   dc.b 111, 112,115, 33,33,0                ;ASCII Codes
   ds 0                              ;null byte
   ...
   public _main
_main:                                       ;here we start
```

When initializing within a function, things look different:

```
/**************************/
/* C code                 */
/**************************/

proc()
{
   char *string = "The Amiga is tops!!";
   ...
}


/**************************/
/* Assembly language      */
/**************************/

   public _proc              ;procedure is 'global'
_proc:
   link a5,#.2               ;allocate stack storage
   movem.l .8,-(sp)          ;save registers
```

```
   lea .1,a0                  ;address of the string -> a0
   move.l a0,-4(a5)           ;a0 -> variable '_string'
   ...
   movem.l (sp)+,.8
   unlk a5
   rts
.2 equ -4
.3 reg
.1
   dc.b 84,104,101,32,65,109,105,103,97,32,105,115,32,84
   dc.b 111, 112,115, 33,33,0
   ds 0                                        ;null byte
```

As you see, the strings are in an unusual form. The ASCII codes of the characters, not the characters themselves, go into the assembler file (see Section 2.2 for information about the common assembler directives `global`, `public`, `dseg` and `cseg`).

Here we should probably explain the width of the individual variables:

`Char` objects are always one byte wide and therefore have a value range from -128 to 127 or 0-255, if you use "`unsigned char`". `Int` variables occupy two bytes with a value range of -32768 to 32767 or 0-65535. `Long` objects are four bytes wide with a value range of -2147483648 to 2147483647 or 0-4294967296. `Float` variables contain four bytes and `double` variables 8 bytes.

Pointers are variables which contain the starting address of a variable or structure. Because the 68000 can theoretically address four gigabytes, pointers must have a range of 0 to 4294967296. Therefore a pointer always occupies four bytes, regardless of what type they point to.

If you want to increment or decrement a pointer to a `long` object, you should note that four bytes are added to or subtracted from the old address. The pointer points to the next `long` object. The same applies to pointers to structures. Assuming a structure holds 95 bytes, then incrementing a pointer to such a structure with `<pointer>++` or `<pointer>=<pointer> + 1` increments the address by 95.

All this time you have assumed that your program only declares and defines variables outside functions. But you can also define variables within a function which are available only for that function. Take a look at the following program:

```
int i;
main () {...}
prroc()
{
   int i;
   ...
   i = ...;
}
```

The variable `i` appears twice. If you access `i` in any line of the `prroc` function, the compiler knows that you mean the `i` defined within this routine. The compiler needs a second symbol table to recognize this. If the compiler wrote the symbol code for the variable `i` in the global symbol table, then it is unable to distinguish between the global `i` defined before `main()` and the `i` in `proc()`.

The local symbol table is used whenever variables are declared and used within a function. When the function is compiled, the next function can access the entire memory range of the local symbol table. When a variable is accessed in the routine, the compiler looks for the variable in the local symbol table first. If the variable could not be found there, then the compiler searches the global symbol table. When using libraries, defining the library base pointer inside or outside the function is important. For example, if you define the pointer to the Intuition library (`struct IntuitionBase *IntuitionBase`) within a function (e.g., `main()`), then you cannot use the Intuition functions.

They require the variable `_IntuitionBase`, which can only be made available for other modules and linker libraries with the assembler directive `global _IntuitionBase,4`. If `IntuitionBase` is defined within a function, then only stack space is reserved for this pointer, which cannot be accessed with the label `_IntuitionBase`.

But back to the local symbol table. Naturally it also needs space in memory. The compiler automatically allocates 1040 bytes for it. If the compiler gives you the error message `Local table full (Use -L)`, you should follow its advice and increase the memory for the local symbol table with the `-L` option. You should note that an entry is 26 bytes long. `-L40` reserves 1040 bytes.

The global symbol table can also be saved—the local symbol table cannot. This is especially useful when you write programs which use many include files.

You can separately compile the include file block (which may contain only include files) and then store the symbol table by using the `+H` option (`cc +Hcompiler_includes include_block.h`) whereby the include block looks something like:

```
#include "exec/types.h"
#include "graphics/gfx.h"
...
#include "graphics/gels.h"
```

Afterward you can read this symbol table back in with the `+I` option (`cc +Icompiled_includes program.c`). This saves the time of reading the individual include files and constructing the symbol table.

2.1.2 Software organization of variables and structures

The contents of variables and structures must be stored in memory locations. The functions request the required memory space from the system stack if no other storage class (`extern`, `static`, `auto`, `register`) is specified for the variable. To do this, the compiler first uses the symbol table to calculate the number of bytes which must be reserved.

Since the compiler knows which variables are being used, and it can find out exactly how much memory is required by each, this calculation is simple. The compiler also accesses the symbol table extensively when the `sizeof()` instruction is used.

But it can lead to serious problems if the variables require more space than the stack has available. You can enable the stack checking with the +m option. Before each routine executes, the routine `_stkchk#()` is called, which calls the routine `_stkover()` if a stack overflow is detected. On a stack overflow, `_stkover()` immediately halts the program and sometimes displays the reset requester (`Finish all disk activity...`).

According to the Aztec manual, `_stkover()` supposedly informs the user of a stack overflow. Unfortunately it doesn't always work that way. Sometimes the guru appears. But you can write your own `_stkover()` routine. This routine should be written in machine language so that it can use an alternate stack. It should be noted that address register a7, which is the stack pointer, points to the highest address of the alternate stack area, since a stack "grows" down.

When linking this new `_stkover()` routine, you should note that a routine by the same name is already stored in the `c.lib`. The linker also has a second machine language routine by the name of `__stkover`: available—the first is yours and the second is that from `c.lib` (an underscore precedes the name of the C procedure in assembly language). The linker uses your routine without the use of a linker option, however, since it appears before that in `c.lib`.

You check for stack overflow only in the development phases. Programs which have to call `_stkchk#()` every time they enter a routine become slower and longer.

Now we come to variable organization in assembly language. Here is a short C program:

```
/*********************/
/* C language        */
/*********************/
int c;
main()
{
   int j;
   j = 0;
   ...;
}

proc()
{
   int i;
   c = 0;
   ...;
}
```

For the sake of simplicity, we are using only `int` variables here. But the principle is the same for allocating or reserving memory for structures, arrays, etc.

Let's look at the assembly source created by the compiler:

```
;:ts = 8                    ;senseless!?!
   global _c,2              ;tells the assembler to reserve
                            ;two bytes for _c. Same as
                            ;"_c: ds.b 2"
   public _main             ;_main is known to all modules
                            ;"as is _c"
_main:
   link     a5,#.2          ;a5 to -(sp), sp to a5, sp + #.2
   movem.l  .3,-(sp)        ;registers on stack (no register
                            ;list specified!) variable stack
   clr.w -2(a5)             ;access variable j
   ...
.4
   movem.l (sp)+,.3         ;saved register contents from stack
                            ;back to register
   unlnk   a5               ;a5 to sp,-(sp) to a5
   rts

.2 equ -2                   ;2 bytes for int object
.3 reg                      ;registers to be saved (list empty)
   public _proc             ;'proc' is known to all modules.
   link a5,#.9              ;allocate variable stack
   movem.l .10,-(sp)        ;save registers
   clr.w _c                 ;c = 0
   ...
.11
   movem.l (sp)+,.10
   unlnk a5
   rts
.9 equ -2                    ;2 bytes for i
.10 reg                      ;no registers to be saved
   public .begin
   dseg                      ;start of the data segment
   end
```

The variable storage for variables in functions comes from the stack, while the storage for global variables, which are defined outside the functions, occupies "normal" memory. These declarations look similar for structures and arrays, except more bytes are reserved. For `unions`, sufficient storage is reserved for the largest variant in the `union`. This memory space is then accessed when a `union` element is accessed:

```
/*********************/
/* C language        */
/*********************/

union num
{
   int i;
   float f;
}

union num z;

main()
{
   z.f = 1.1;
   ...
   z.i = 10;
   ...
}
```

Let's look at the assembly source created by the compiler:

```
   global _z,4                ;reserve bytes for the union
                              ;(global)
   public _main
_main:
   link a5,#.2
   movem.l .3,-(sp)
   move.l #$8ccccd41,_z       ;z.f = 1.1 (float
representation);
   ...
   move.w #10,_z              ;z.i = 0
   ...
   movem.l -(sp),.3
   unlk a5
   rts
.2 equ 0                      ;no local variables
.3 reg
   public .begin
   dseg
   end
```

When accessing a structure or array element, the offset of the element in question must be computed in addition to the base address of the structure or array:

```
/*********************/
/* C language        */
/*********************/

main()
{
struct Structure
   {
      int Element1;
      long Element2;
      float Element3;
   }

struct Structure s;
struct Structure t;

   s.Element1 = 0;
   ...
   s.Element2 = s.Element3;
   ...
   s=t;
}
```

Let's look at the assembly source created by the compiler:

```
   public _main
_main:
   link a5,#.2
   movem.l .3,-(sp)
   clr.w  -10(a5)                    ;s.Element1 = 0
   ...
   move.l -4(a5),d0                  ;s.Element3 -> d0
   jsr Ffix#                         ;convert float to long
   move.l do,-8(a5)                  ;d0 -> s.Element2
   ...
   lea -10(a5),a0                    ;&s to a0
   lea -20(a5),a1                    ;&t to a1
   move.l (a1)+,(a0)+                ;convert bytes
   move.l (a1)+,(a0)+
   move.w (a1)+,(a0)+
.4
   movem.l (sp)+,.3
   unlk a5
   rts
.2 equ -20
.3 reg
   public .begin
   dseg
   end
```

As you can see, this program uses variables of type `float`. It must therefore be compiled with the `+fi` options. When using `double` variables, the `+ff` option must be used. When using the 68881 math coprocessor you must compile your programs with the `+f8` option. In all three cases math routines are needed which are not in the `c.lib`

library, therefore m.lib, ma.lib or mx.lib must also be linked in with the object file.

Now we'll look at the memory classes we mentioned earlier: With the memory class extern you are informing the compiler that the variable is found in a different module. The compiler must not reserve any space for this variable here since this must be done in another module. If this does not happen, the linker displays an unresolved reference error.

If you use the extern class in a variable declaration in a function, the global symbol table searches for the variable. If you want to use a global variable, you should always specify the storage class extern in the declaration. This way you always know which variables in the function are being used; which are local and which are global:

```
int i;
proc()
{
  extern int i;     /* access to global i (declaration) */
         int j;          /* local (auto) j (definition) */
  ...
}
```

The static storage class gives the compiler information about the type of storage which should be used for the variable. Normally all of the variables in a function are auto. This means that memory space is allocated on the stack when the function is entered, and then released when control leaves it. If you declare a variable static, then the memory space is taken from the normal program storage. When the function is left, the value is retained so that you can use the old value when the function is re-entered. The bss directive of the assembler is used to assign static storage space. This is similar to the global directive, except that the storage space for the routine in which the static variable is defined is valid. This results because the variable is not accessed by its name, as in global_c,2, but with a normal label (like .4).

static variables are particularly useful in recursive functions. They can be used as counters or to quickly pass arguments without having to pass them to the function every time (see Section 2.1.4).

With the register storage class, a CPU register is used to hold the value. You can declare any scalar variable type as a register variable. Note that double variables which are declared as register must have some bytes truncated. double variables are eight bytes wide, but the address and data registers of the 68000 can hold only four bytes each.

The registers D0, D1, D2, D3, A2 and A3 can be used by the compiler for storing values.

2.1.3 Translating control statements

Let's look at control statements. They determine the order in which various routines and commands are processed.

After the error search has produced no (apparent) errors and all variables and structures have been defined and allocated, the actual compiler process begins, translating the individual C instructions and function calls into assembly language. First we'll compile the control structures (`for`, `while`, `switch`, `if`):

```
"for (i=0;i<5;i++) j += 2;".
```

Compiled, this program segment looks like:

```
        clr.l -4(a5)     ; i=0 (intialization of
                         ;      'i')

.1:     add.l #2,-8(a5) ; j += 2 (loop body)

        add.l #1,-4(a5) ; i++     (increment)
        cmp #5,-4(a5)   ; i<5     (termination condition)
        blt .1          ; yes, then back to the beginning
```

Once the compiler recognizes this instruction, the initialization of the variable `i` is checked. The compiler determines that the variable `i` is being set to 0. This is represented by the assembly language instruction `clr.l -4(a5)`. It is assumed that the storage space for the variable `i` is reserved in `-4(a5)` (stack storage). The variable `j` is located at `-8(a5)`.

After the initialization of the variable `i` the compiler must then look at the loop body (`j += 2`). The loop condition and incrementing of the variable `i` are ignored for the moment. The compiler goes to the statement `j += 2`. Since this is the first (and only) statement in the loop body, a label must precede this statement so that you can jump to it. After the label is set, the actual statement is translated (`add.l #2,8(a5)`). Now the compiler takes care of incrementing the variable `i` (`add.l #1,-4(a5)`) and the loop condition (`cmp.l #5,-4 (a5); blt .1`).

Here you can see that the linearity of the C commands is retained in assembly language, only in reverse order.

The `while` loop compiles in a manner similar to the `for` loop. In a `while` loop, the condition specified immediately after the keyword `while` is translated immediately. The loop condition is tested before the loop body executes. The compiler notes that it must set a label at the end of the `while` body so that if the `while` condition is not fulfilled, it can jump past the body:

```
/*******************/
/* C language      */
/*******************/

  i = 0;                              /* Have to do the */
                                   /* initialization ourselves */
  while (i<5)                        /* while loop             */
     {
       j += 2;                       /* loop body              */
       i ++;
     }
```

Here is the assembly source created by the compiler:

```
        clr.l -4(a5)        * i = 0;

.1:     cmp #5,-4(a5)       * i<5
        bge .2              * no
        add.l #2,-8(a5)     * { j += 2;
        add.l #1,-4(a5)     *   i ++; }
        bra .1:             * back to the start of the loop
.2:     ...
```

There is a second type of `while` loop:

```
/*********************/
/* C language        */
/*********************/

i = 0;
do {
     j+=2;
     i++;
   } while (i<5);
```

In this type of `while` loop the body executes at least once. The loop condition is checked after each execution.

Another control structure is the `switch` statement. It provides a way of choosing from several alternatives. A typical `switch` statement looks like this:

```
switch (i)
  {
     case 3:
        i++;
     break;

     case 4:
        i--;
     break;
```

```
    default:
      i=0;
    break;
}
```

The compiler turns this into:

```
    move.l -4(a5),d0          * contents of "i" to d0
    bra .4

.6: add.l #1,-4(a5)           * case 3: i++;
    bra .5                    * break;

.7: sub.l #1,-4(a5)           * case 4: i--;
    bra .5                    * break;

.8: clr.l -4(a5)              * default: i=0;
    bra .5

.4: sub.l #3,d0
    beq .6
    sub.l #1,d0
    beq .7
    bra .8
.5: ...
```

First the compiler moves the value of the variable `i` to data register `d0`. Each `case` portion is then translated, whereby a label is first placed before each portion and then the individual statements are translated. The use of the `break` statements at the close of each `case` portion causes a branch to the instruction after the switch.

How is the `switch` statement processed by the compiler? It first causes the variable `i` to be loaded into data register `d0` and then the branch to `.4`.

When the compiler reads `case 3`, the value 3 is subtracted from `d0` and the result tested for 0. A branch is made if true. In the second case the value is tested against 4. But instead of loading the value of the variable `i` back into `d0` and then subtracting 4, the compiler calculates the difference between the first and second `case` (3-4). If this difference is negative, then the difference is subtracted from `d0` (`sub.l #1,d0`). If the difference is positive, it is added (`add.l #1,d0`). Then you can again test the data register against 0.

If all of the `case` comparisons fail, the default statements are automatically executed (the default is optional, in which case the `switch` statement is simply exited).

But naturally a `switch` statement is not always as ordered as was shown above. It may contain more than two `cases`, and they do not have to be ordered.

```
switch (i)
  {
     case 1:
        i++;
     break;
          case 0:
        i--;
     break;
     case 6:
        i=0;
     break;

     case 2:
        i=0;
    break;
  }
```

We will ignore the fact that this isn't exceptionally elegant programming—the compiler doesn't mind. What does it do? The following assembly language code clarifies this:

```
     move.l -4(a5),d0          * contents of 'i' to d0
     bra .4

.6:  add.l #1,-4(a5)           * case 1: i++;
     bra .5                    * break;

.7:  sub.l #1,-4(a5)           * case 0: i--;
     bra .5                    * break;

.8:  sub.l #1,-4(a5)           * case 6: i--;
     bra .5                    * break;

.9:  sub.l #1,-4(a5)           * case 2: i--;
     bra .5                    * break;

.10: dc.w .7-.11-2             * case 0:     (0)
     dc.w .6-.11-2             * case 1:     (2)
     dc.w .9-.11-2             * case 2:     (4)
     dc.w .5-.11-2             * default:
     dc.w .5-.11-2             * default:
     dc.w .5-.11-2             * default:
     dc.w .6-.11-2             * case 6:     (12)
    .4:  cmp.l #7,d0
     bcc .5                    * default
     asl.l #1,d0               * d0 * 2
     move.w .10(pc,d0.w),d0
.11:
     jmp (pc,d0.w)
   .5:  ...
```

This `switch` statement is translated similar to the first. Only the individual `case` selection is handled somewhat differently.

First the `case` blocks are translated in the order in which they appear in the C program. They are then placed in order in the jump table which contains the addresses (relative to the PC) of all of the `case` blocks. The offsets are then sorted according to the order of the `case`s to which they point and are stored.

Under normal circumstances you can test a maximum of 100 `case` conditions. If you want to test more cases–which seems rather unlikely, the `-Y` option allocates more space for additional `case` conditions. `-Y100` reserves 100 entries for the `case` conditions. Each entry consists of four bytes, whereby two bytes are used for the offset and the other two bytes for the integer number for which the case is tested.

When selecting a `case` condition the value of the variable `i` (contained in `d0`) is doubled (left-shifted) and used an index into the jump table. Since this table consists only of offsets which are specified in 16-bit words, `i` only has to be doubled instead of quadrupled, as would be necessary if the table consisted of the absolute addresses of the `case` blocks.

Since some cases are not tested (`case 3:`, `case 4:`, `case 5:`), the jump table is filled at these locations with the address of the command after the `switch` statement (label .5).

If the cases to be tested are far apart from each other, then they are not selected with a jump table. This would not be an efficient use of memory because the default label would appear many times in the jump table. Therefore the first method is used here—the variable to be tested is placed in `d0` and the various `case` values are subtracted or added and `d0` is tested against 0.

Now we come to the simplest control structure: The `if/else` statement. These can be translated linearly, as they are written in C:

```
if (i == 1) i = 0;
else
   i++;
```

In assembly language this looks like:

```
     cmp.l #1,-4(a5)        * i == 1  ?
     bne .4                 * no
     clr.l -4(a5)           * yes
     bra .5                 * continue
.4:  add.l #1,-4(a5)        * else i++;
.5:  ...
```

The label poses the only problem when translating the `if` statement. The compiler notes which label number it must use after the branch command, for example, and which label must be placed before a command.

Another problem is the evaluation of mathematical expressions. What does the compiler do with `i = (j*5)+20`? The compiler starts with the deepest level of parentheses and works it way up.

To describe this process in detail would be too complex here. If you aren't sure which operations have priority over others, you can add more parentheses. This ensures the portability of your program, even if other compilers translate mathematical expressions somewhat differently.

Another control statement is the `goto` statement, although few structure-conscious C programmers use it:

```
goto Label;
...
...
Label: ...
```

The assembler source generates a `bra` or `jmp` command. The compiler also must ensure that the label being jumped to is in the same function as the `goto` statement.

2.1.4 Function calls

In addition to using control statements and variable assignments, you can also write your own functions or use existing ones. There are two types of functions: those which return a value, and those which do not. Both types let you pass arguments to them. Arguments are passed through the stack.

```
func(i,j,k)
int  i;
long   j;
struct nonsense *k;
{
   int l;
   l=0;     func (i,k,k)
}
```

In assembly language, this somewhat senseless recursive function looks like this:

```
_func:
   link a5.#.2              ;storage for local variables
   movem.l .3-(sp)          ;registers on stack
   move.l 14(a5),-(sp)      ;parameter k on stack
                            ; for subsequent call
   move.l 10(a5),-(sp)      ;j on stack
   move.l  8(a5),-(sp)      ;i on stack
```

```
   jsr _func                    ;call routine
      lea 10(sp),sp             ;restore stack
      movem.l (sp)+,.3          ;load registers again
   unlnk a5                     ;release local storage
   rts

.2 equ -2                       ;two bytes for int object
.3 reg                          ;register list
```

The following figure illustrates how the local variables are placed on the stack:

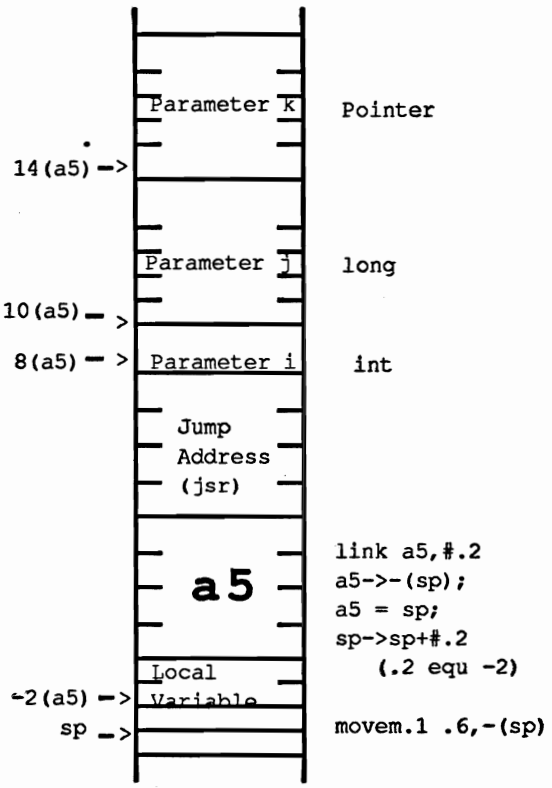


Figure 2.1

You can see that the input arguments are accessed with positive indexing on a5. (e.g., 8(a5)), while negative indexing accesses local variables (-2(a5)).

When the input arguments are accessed, the old value of register a5, placed on the stack by the link command, and the return address, placed on the stack by any JSR, must be skipped. Beyond this, the input parameters are simply accessed in reverse order on the stack.

The compiler doesn't check if the correct number of arguments were passed to a function.

Let's assume that a function expects three `long` variables as input arguments, but you pass this routine three `int` values instead. The three `int` variables occupy only six stack bytes, while the routine accesses 12 bytes (3*4 (`long`)).

If you now change these parameters (in the above example, k=0 <=> "`clr.l 14(a5)`"), which is possible in C, you may end up changing a return address because when the arguments were passed, only six bytes were allocated on the stack instead of 12.

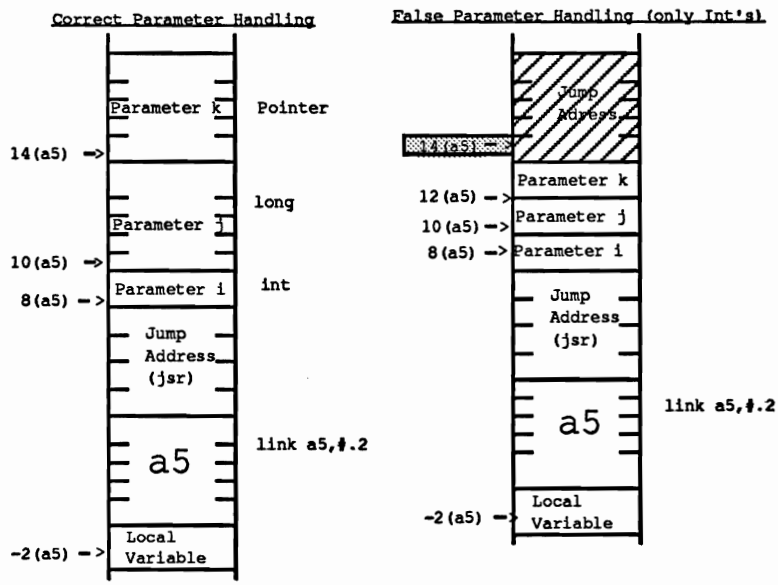


Figure 2.2

For functions which return a value, you must specify the type of the return value. This is given immediately before the function name. There are also functions of type `void`, which means that they do not return a value. If you do not explicitly set the return type of a function, the compiler assumes that it is `int`.

The type of the function's return value is stored in the symbol table just like a variable type:

```
void Function (..., ..., ...)

Symbol table:
-------------
"Function"/code_function/code_void
```

If a function is in another module or in a library, the type can be declared with `extern` like an external variable:

```
extern struct *MsgPort FindPort();

Symbol table:
-------------
"FindPort"/code_external_function/code_structure_type/
"MsgPort"
```

You can now be sure that the compiler will not "complain" when you assign the return value of `FindPort()`, which is a pointer to a `MsgPort` structure, to a pointer of type `*MsgPort`. Basically, the objects of the pointers don't matter when doing pointer assignments. Pointer variables are always four bytes long, so that no conversions have to be made. There is a second type of pointer, however, the `BCPL` or `CPTR` pointers. These are leftovers from the BCPL language. Their contents are shifted right by two bits so that they can access only long words. `BCPL` pointers occur only as input parameters, and then only rarely, such as in certain I/O routines.

You can also use `extern` to assign different return values to functions in other modules. Let's say that the routine `atol()` is defined in module 2 and that it actually returns a `long` value. If you now define in module 1 that this routine returns a value of `int` type (`extern int atol()`), the compiler won't be any the wiser because the symbol table for the second module is not available. Note that this declaration must take place before the function is used. Even when you write your own functions of type other than `void`, you must ensure that they are defined (or declared before they are used) so that they are entered in the symbol table.

For functions, the input argument types are irrelevant. There are compilers, however, which check the input parameters for "type purity" so that errors like those described above don't occur. The Aztec compiler is not one of these.

2.2 The Assembler

After the compiler has created the assembly language source, the assembler can begin its work. It must create the object file from the assembly source. Its call is as follows:

```
as [>output_file] [options] program.asm
```

The object file consists of executable machine code whenever possible. Only unresolved external references have to be resolved by the linker.

Let's look at the following assembler source:

```
   dseg                          ;data segment
   ds 0                          ;reserve 0 bytes
   public _Str                   ;global variable
_Str:
   dc.l .1+0          ;allocate memory with address of the
   cseg                          ;code segment
   dseg                          ;data segment
.1    dc.b 72,97,108108,111,0    ;"Hello"
   cseg                          ;code segment
   global _i,2
   public _main
_main:                           ;start here
   link     a5,#.3
   movem.l  .4,-(sp)
   add.l 1,_str                  ;str++;
   jsr _exit
.5
   movem.l (sp)+,.4
   unlk   a5
   rts

.3 equ 0
.4 reg

public _exit
public .begin
dseg
end
```

From this assembly language source the assembler creates relocatible code. Relocatible means that the program is executable at any location in memory. It is not restricted to a particular address.

But the assembly language source is not set up such that it creates relocatible code. Such code uses PC-relative mode addressing (e.g., `jmp $10(pc)`). The assembler source contains only instructions with absolute addressing (e.g., `jsr_exit`).

During assembly, these absolute instructions are converted to relative instructions. The directives `public` and `global` play a role in this. They enter the specified labels in the symbol table which contains all of the global labels in the module.

In the first pass the assembler searches the program for such labels. At the same time it checks to see if a label in the symbol table corresponds to something in the source file. This is the case with `main`, for example. First this label is placed in the symbol table (`public _main`) and then defined (`_main:`). `global _i,2` places the label `_i` in the symbol table. The memory storage for these variables is not made available until the program is linked. The `bss` directive also reserves storage for a variable. The label associated with the storage is not placed in the symbol table, however. Therefore `bss` is well suited for declaring static variables within functions.

When an instruction now accesses a memory location specified by a label, the symbol table is searched to see if this label is defined in the same module. If this is the case, the offset from the current position to the label can be calculated for the instruction.

When calculating the offset, it is important to know whether the label describes a memory location in the code segment or the data segment. From the first pass through the program the assembler knows how many bytes the program contains. If an instruction accesses a label in the data segment, you must take into account the fact that the data segment is appended to the code segment.

The code and data segments are two distinct areas. If the directive `dseg` stands before an instruction in the assemble language source, then all subsequent data is written into the data segment until a `cseg` occurs, marking the start of the code segment.

In the following listing you can clearly see that the code and data segments are two separate areas. Whenever the directive `dseg` or `cseg` appears, the address in the second column changes (the code segment/ data segment program counter).

```
Aztec 68000 Assembler 3.4a 1-25-87
     1 0000:                        dseg
     2 0000:                        ds 0
     3 0000:                        public _str
     4 0000:                    str:
     5 0000:  xxxx xxxx             dc.l .1+0
     6 0004:                        cseg
     7 0000:                        dseg
     8 0004:                    .1
     9 0004: 4861 6c6c 6f00         dc.b  72,97,108,108, 111,0
    10 000a:                        cseg
    11 0000:                        global _i,2
    12 0000:                        public _main
    13 0000:                    _main:
    14 0000: 4e55 0000              link a5,#.3
    15 0004:                        movem.l .4,-(sp)
```

```
16 0004: 52ad xxxx          add.l #1,_str
17 0008: 4eba xxxx          jsr _exit
18 000c:                 .5
19 000c:                    movem.l (sp)+,.4
20 000c: 4e5d               unlk a5
21 000e: 4e75               rts
22 0010: 0000 0000       .3 equ 0
23 0010: 0000            .4 reg
24 0010:                    public _exit
25 000c:                    public .begin
26 000c:                    dseg
27 000a:                    end
```

This listing is obtained by using the `-l` option of the assembler. Here you can see the assembler changing absolute addressed commands into PC-relative instructions. The machine code for `jmp_exit` would be $4eb9. In the listing (line 17), the code for `jmp_exit` is $4eba. This stands for `jsr XXXX(pc)`. The offset of the `_exit` routine cannot been determined yet because this routine is not defined in this module.

The first pass of the assembler creates this listing. This can be seen by the fact that line 5 is full of x's. This happens because in the first pass it isn't known where the string resides in the data segment. Similarly in line 16 (`add.l #1,_str`). Since the start of the data segment has not yet been determined, the offset to the `_str` variable cannot be determined.

The object file which the compiler creates consists of executable machine code whenever possible; a header which specifies the size of the program; the number of labels in the symbol table, etc.; and the symbol table itself:

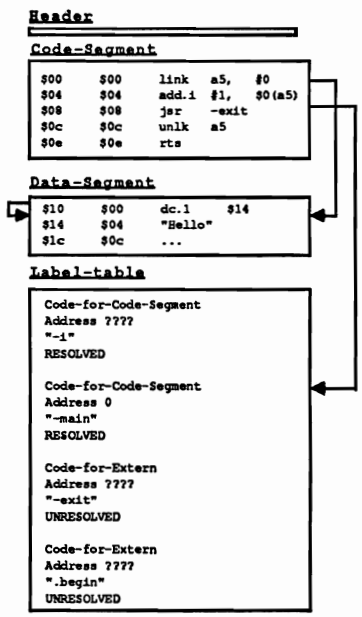


Figure 2.3

Let's look at the listing and the assembly language source. When you compare the two, it appears that the assembler is not assembling the instructions `movem.l .4-(sp)` (line 15) and `movem.l -(sp),.4` (line 19).

The assembler "optimized away" these instructions because they are clearly senseless. The instructions are useful only when there are registers to save. This is only the case when `register` variables are used, however.

The assembler also removes branches to the next command:

```
brs .7
.7 ...
```

This construction is also useless since the instruction at label .7 would execute without the branch command. Also, all `jsr` instructions are replaced with `bsr`, if possible, which executes faster.

Other assembler options exist in addition to the `-n` option:

Assembler options

`-O filename` The object code is written to the specified file.

`-Idirectory` This option tells the compiler which directory to search for include files.

`-L` This option produces a listing of the assembly source. The listing is created in the first pass of the assembler, in the file `<filename>.lst`.

`-N` This option disables object code optimization.

`-Snumber` This option creates a squeeze table with numeric entries. An entry is one byte long. The assembler allocates a default of 1000 bytes (entries) if the `number` argument is omitted. The squeeze table is required for the optimization of the object code.

`-V` This (Verbose) option prints memory statistics.

`-ZAP` This option deletes the source file after assembly.

`-C` This option creates "large code" (see Section 2.3).

`-D` This option creates "large data."

`-Ename[=value]` This option assigns the argument `name` to `value`. This assignment is the equivalent of the `name EQU value`. The `value` argument is optional, and if not specified, the symbol is assigned the value 1.

2.3 The linker

The linker resolves unresolved references in the individual modules. To do this, the object files (modules) are usually linked to the linker libraries. The call is as follows:

```
ln [>output_file] object_file.o [options]
```

Let's look at the last example program, which called the `_exit` routine. This routine wasn't defined in the module in which it was called, however. The `_exit` definition takes place in the linker library `c.lib`. This must be linked to the object file which contains the `_exit` call.

The linker first reads the object file. It looks at the symbol table. From the symbol table the compiler learns that `_exit` is an unresolved external reference. The linker notes this in a table which lists all of the unre-solved references. If one of these references is later resolved in a module, then this is noted in the table.

Now the linker reads each module in order. Up to now, the commands which called external routines, for example, were not complete. The hex code for the PC-relative `jsr` instruction exists, but the offset of the routine to jump to is unknown. These offsets can be calculated in a second pass and then entered at the appropriate location.

Let's look at the following assembler source:

```
   dseg                          ;data segment
   ds 0                          ;reserve 0 bytes
   public _Str                   ;global variable
_Str:
   dc.l .1+0          ;allocate memory with address of the string
   cseg                          ;code segment
   dseg                          ;data segment
.1    dc.b 72,97,108108,111,0    ;"Hello"
   cseg                          ;code segment
   global _i,2
   public _main
_main:                           ;start here
   link     a5,#.3
   movem.l  .4,-(sp)
   add.l 1,_str                  ;str++;
   jsr _exit
.5
   movem.l (sp)+,.4
   unlk   a5
   rts

.3 equ 0
.4 reg
```

```
public _exit
public .begin
dseg
end
```

In line 17 you see the instruction `jsr_exit`. The assembler generated the hex code `$4eba xxxx` for this instruction. `jsr_exit` actually occupies four bytes. But the assembler uses only the first two bytes—namely those used for the code for the PC-relative `jsr` instruction itself (`$4eba`). The remaining two bytes, which specify the offset for this instruction, remain initialized. This initialization is handled by the linker, which can calculate all of the necessary offsets in a second pass and enter them in the appropriate locations.

Now we come to a question you are probably asking yourself: When you link an object file with a linker library, are all of the functions of the library linked in or just the functions actually used by the object file? Here the Aztec linker is very economical. It appends to your program only the library routines actually used, when linking two object files, it combines all of the functions.

But here again we have a question: Since you can link `c.lib` to a module with `ln Ofile.o -lram:c` as well as `ln Ofile.o ram:c.lib`, and since `c.lib` is just an object file which contains many functions, how are only the needed routines from `c.lib` added to the `Ofile` and not the entire `c.lib`?

The answer is that the linker libraries are not object files in the sense of the files which are created by the assembler. The linker libraries consist of a combination of several normal object files. These libraries are organized in a somewhat different manner. While the symbol table is placed at the end of the file in normal object files, the table moves to the beginning of a linker library. Then comes the executable machine code.

If the linker is to link a library to an object file, which is almost always the case, it recognizes the library by the special code at the beginning of the file. The linker then reads the library symbol table, which it uses to immediately resolve external references in the object file. It also notes which library routines are used.

When the linker has read the symbol table, it then comes to the executable code of the library. It removes the functions that it noted and appends them to the normal object files. In the second pass it calculates all of the necessary offsets and sets them.

The linker is not only responsible for resolving unresolved external references. It also links startup code to the object file. This is accomplished by the assembler directive `public .begin`. `.begin` is stored in the list of unresolved references in the first pass and then appended to the program later, when linking with `c.lib`.

In addition, the machine language instruction `jmp .begin` (in PC-relative form) is created for each module which contains the directive `public .begin`. This is necessary only for the first object file to be linked. This command must be the first one in the program. In the other modules this command just takes up space (four bytes). Therefore when you use multiple modules you should make sure that the label `.begin` appears only in the first module. You should compile all other modules with the `+b` option.

Now we come to the startup code. First it opens the DOS and `MathFFP` libraries. Then the startup code determines whether the program was started from the Workbench or the CLI.

When starting from the Workbench, it must first wait for the startup message. If this message passes arguments, they are interpreted as a `Lock` structure and designate the `Locked` directory as the current directory. The `Tooltype` window opens, if necessary, and defaults to standard input/output.

When starting from the CLI, the startup code ensures that the program gets the number of arguments and the address of the argument array. These are made available by the DOS through the CIS (Command Input String). The CLI's startup code prepares the parameters so that they can be passed through `main(argc, argv)`.

After processing the parameters string, the `main` routine of the C program is called. Somewhere in the startup code, which consists partly of machine language and partly of C code, the call `main(argc,argv)` appears. Directly after this is the C call `_exit(0);`. This `_exit()` ensures that the DOS and `MathFFP` libraries are closed, that the memory space allocated in connection with preparing the command parameters is released and the command `_exit()` is called. This `_exit()` is precisely the routine which your program can call to terminate the program prematurely.

But let's return to the linker libraries. Since the startup code must always be linked to a C program, `c.lib` must always be part of your program. There are other libraries, however, such as `m.lib`. This library must always be linked to an object file if floating-point arithmetic is used. It isn't enough to simply link this library to the object file. You must also compile the program with the `+ff` compiler option. To use double precision floating point arithmetic, you must compile the program with the `+fi` option and link with the `mx.lib` library. When using the 68881 floating-point coprocessor you must use `m8.lib` and compile the program with `+f8`.

But even `c.lib` and `m.lib` are not the only libraries in the compiler package. If you have ever looked around on the disks, you would have found a `c32.lib`, a `cl.lib` and a `cl32.lib`. The other libraries also appears in these forms: `ml.lib`, `ml32.lib`, etc.

The linker libraries with a 1 in their names must be linked to the object files when you want to use the "large data" and "large code" memory model.

2.3.1 "Large data" and "small data"

Now we come to the differences between "large data" and "small data". As the name implies, these two memory models refer to the data segment. In the "small data" model, the data segment can be only 64K in size. This is because access to the data is performed with an address register which points to the middle of the data area and through positive and negative offsets, between 0 and 32768, can only access 64K bytes. Here the PC-relative addressing with 16-bit offset comes into play.

With the "large data" model, data are accessed by their absolute addresses. But since it is unknown from the beginning where the program resides in memory, all of the instructions which access absolute memory locations in the data segment must be corrected. This is the reason that a "large data" program takes longer before it starts to run. Absolute address access is also slower than PC-relative access to memory locations. The advantage of the "large model" is that the data segment can theoretically occupy the entire free memory.

2.3.2 "Large code" and "small code"

The models "large code" and "small code" refer to the code segment of the program. It doesn't involve the executable code so much as the data which are also stored in the code segment.

With the "large code" model data can reside in the entire code segment, even scattered throughout memory. The "small code" model, on the other hand, allows data to reside only in 32K of the program counter.

If the "small code" program is larger, an address table is created with absolute addresses through which the outlying memory areas can be accessed indirectly.

Also, a jump table is used for branches outside this 32K. Correcting this jump and address table normally takes less time then correcting the instructions in a "large code" program, which uses absolute addresses exclusively. With the "small code" model an address register must be set to the middle point of the data segment, while "large code" accesses the data segment absolutely. The absolute 32-bit addressing is also slower than the 16-bit offset addressing.

If you want to use either "large code" or "large data," you must compile and assemble the program with the +C or +D option. In both cases you must link in a library with an `l` in the name (such as `cl.lib` or `cl32.lib`).

Now you may be wondering what the "32" means in names like `c32.lib` and `cl32.lib`. This 32 stands for 32-bit integers. The `int` variables used in these libraries are 32 bits in length instead of 16.

This is especially important for compatibility between Lattice and Aztec programs. Lattice programs don't use 16-bit `int` variables. The compiler extends variables and constants to 32 bits. This extension is not performed by the Aztec compiler when using an `int` variable as a function parameter, 16 bits are passed instead of 32. The Amiga library routines require 32 bits, however, and the arguments may be interpreted incorrectly, or the system may even crash.

The Aztec compiler can emulate the Lattice compiler when you extend all `int` variables and constants to 32 bits. This is accomplished with the +L compiler option.

The C support functions in `c32.lib`, `cl32.lib`, etc., were compiled with this +L option and are useful when compiling Lattice source codes under Aztec.

Here is a list of the linker options:

Linker options

`-O filename` The executable program is stored as `filename`. If this option is omitted, the executable program retains the object file's name without the extension of `.o`.

`-Lfilename` This option instructs the linker library to be linked to the object file.

`-F filename` The linker reads the arguments from the ASCII file specified here. This is useful if you want to pass a large number of arguments to the linker.

`-T` This option stores the linker symbol table as an ASCII file. The name of this ASCII file is the object filename with an extension of `.sym`. This ASCII file contains information about the file—in which the segment code is located. This file contains "hunks" of executable instructions, system variables and program variables as well. These hunks are enclosed in the labels `__Hx_org` and `__Hx_end`, whereby `x` represents the given hunk number. In programs which consist of only one module, the program variables are stored between

__H2_org and __H2_end. The program begins at __H0_org.

-W — This option stores information for the db debugger. This information includes the labels and their offsets from the start of the program (see -T above).

-V — This option prints hunk statistics.

+O[i] — This option instructs the linker to write the subsequent object module to the next or specified code segment.

+C[cdb] — This option instructs the linker to load the code segment (+Cc), the data segment (+Cd), or both segments (+Cb) into chip memory.

+F[cdb] — This option instructs the linker to load, the code segment (+Fc), data segment (+Fd) or both (+Fb) into fast memory.

+A — This option instructs the linker to begin each object module on a longword address.

-C, -Q — These options instruct the linker to store information for the source level debugger (not currently implemented).

+Q — This option instructs the linker to suppress the procedures of the object files being linked.

-M — This option prevents user-written linker library functions from overwriting the linker library routine (see stack overflow).

+L — This option instructs the linker to assume that the following object modules are "true" Amiga linker libraries. Attempting to link c32.lib under this option was unsuccessful.

`+s`
`+ss`
`+sss`

The `+s` option instructs the linker to write the executable code of each object module and the linker library routines used by the modules into separate hunks. The `+ss` option instructs the linker to write the code into one hunk, providing the code does not exceed 8K in length (after 8K a new hunk is used). The `+sss` option instructs the linker to use a hunk for each module. You should note that the linker libraries are combinations of multiple modules and so each routine used is written in a separate hunk. Without specifying `+s`, `+ss` and `+sss` the entire code is written into one hunk.

2.4 The debugger

Theoretically your C program should run correctly. But what do you do if the program doesn't run, and you just can't explain why the program doesn't do what you had wanted it to?

The debugger can help you find errors. It monitors executable programs for runtime errors. There are two basic types of errors:

1. Those which return incorrect results
2. Those which cause a Guru Meditation

With the first type of error, you enter the debug mode of the included debugger (`bin` directory of the third disk) with the `al` command. This command waits until a program has been started from the Workbench or the CLI.

After you have started the program the debugger responds immediately. It attempts to load a symbol table (`Loading syms...`). This symbol table is similar to that which you examined for the compiler. Here the table must be created by the linker (`-W` option), which contains all of the labels used as well as their offsets from the start of the program. Therefore in the development phase you should link your program with `-W`. The final and error-free version of the program should not be linked with the `-W` option, especially if you intend to publish the program in any form. This additional information is very useful for others who want to know how your program works.

After the symbol table loads, the debugger displays all of the CPU registers. In addition to the program counter (Pc), status register (Sr) and condition flags (x, n, z, v, c), all of the data and address registers are also displayed. In addition, the debugger disassembles the first instruction of the C program, e.g.:

```
'__HO_org jmp .begin'
```

As mentioned previously, this instruction (`jmp .begin`) must be the first one in every C program, since it handles such tasks as reading the command parameters.

Now you can disassemble additional instructions of the program by selecting `u` (unassemble). You may see instructions like `_main link a5,#-6`, etc. You can thank the symbol table created by the linker and loaded by the debugger that you see the word `_main` instead of a more or less meaningless number.

But being able to see the program doesn't help much when looking for errors. Therefore the debugger gives you something called single-step mode. Each time you select `s`, the current or previously displayed instruction executes. The registers are then redisplayed and the next command is disassembled.

It's very tedious to have to step through the startup code at the beginning of the program. The startup code cannot be omitted, however, since it is here that the command parameters are processed and important libraries are opened. Therefore you should set a breakpoint at the routine `_main` (`bs_main` (breakpoint set)). The `_main` routine is called at the end of `.begin`.

After setting the breakpoint you can start the program by selecting `g` (go). The program (startup code) executes until a breakpoint is reached. Since this occurs at `_main`, you can proceed through the actual program in the single-step mode, whereby the command parameters fetched by the startup code are also taken into account.

If during single-stepping you come across a call to `_printf`, for example, you don't have to worry, you don't have to go step by step through this long routine. The `t` (trace) command executes this routine and returns to the next instruction in the program.

If you have a copy of the assembler listing augmented by the C statements as comments (`-T` compiler option), you can go through the program step by step, and when you discover an error, you know exactly which C statement is at fault.

Now to debugging a program which gives you a `Finish ALL disk activity...` requester. One possibility would be to debug the program in the manner just described until you can find the error. Or you can debug the task in which the error occurs.

Imagine the following example: A program leads to a division by 0. This displays the requester just mentioned, and you can return to normal operation only by resetting the computer.

But when you start the debugger (through a new CLI) or are in an already running debugger, you may be able to save the system before a total crash, and perhaps save important data (by jumping to a routine which stores the data, for example).

To do this, you must use the `ap` command (debug post mortem). With this command you can debug a task after the `Finish...` requester. You are asked for the number of the task which you want to debug. It would probably be useful to have previously used the `at` command to view the current system tasks. There you might see a line like:

```
3:   (0000a708)        0  Wait      Initial CLI      <div>
```

If you want to debug the task in which the program `div` is running, you must enter the number 3 when you are asked for the task to debug. `ap` suggests which task to debug by noting which task just caused an error. After entering the number the debugger goes to the location in the selected task (program) in which the error occurred. You can then debug the program as before.

Two other commands can be useful when debugging: `dc` and `dg`. `dg` displays the contents of all global variables (display globals), while `dc` specifies at which address a given label is found (display code).

Back to task debugging. In addition to the `ap` method you can also debug any task in the system, not just those which cause a serious error (you can debug such tasks with `ap`). In addition there is the `as` command. Here too, you are asked for the task to be debugged and the debugger tries to load the symbol table for the task. The program must be on the disk in the current directory.

But the first disassembled line of the task is:

```
'fc08b8    rts'
```

This `rts` instruction doesn't belong to the current task itself. It belongs to an essential part of the task handling mechanism. This instruction is located in the section of the operating system which ensures that only one task is running and all other tasks are passive until one of these tasks is activated by a message from the keyboard, mouse, disk, etc.

How do you access the task? You just have to enter the `g` (go) command or the `s` (step) command, and then "click" the task, move or change the size of a window managed by this task.

There are also tasks which wait for a keypress or something similar. The important thing about task debugging is to make sure that a "message" is sent to the task after `s`. You can execute the task without hindrance from the debugger with `ar` (resume).

To get data about the current task, use the `ai` (information) command to display information about the task or its task structure. In addition to the status, you will also see the Trap Allocation and Trap Able mask, whether the task is part of a process (Process Info) or a CLI structure. The contents of the most important structure elements are displayed, whereby the names of the elements can be shortened somewhat ("Rslt" for "Result").

2.5 Tips and tricks

Now that we have explained how the compiler, assembler and linker are used, how they work, and how to find errors, we want to give you a few programming tips to get you on your way.

2.5.1 Accessing absolute memory locations

You have probably wondered at one time or another how to access a specific memory location in C. Remember that pointers store the starting addresses of variables, structures, unions, bit fields, etc. If you can increment pointers (actually the address to which a pointer points), why can't you assign an absolute value to a pointer? The only thing to note is the type of variable to which a pointer points. For example, if you want to read a hardware register, the pointer you use should point to type `uword`, since all hardware registers occupy two bytes. This allows you to read the entire word.

An example which can explain access to absolute memory locations is accessing a `CIAA` register, which occupies one byte. The address of this register is `$bfe001`. With this register you can read the joystick fire buttons and the left mouse button. Bit 7 (2^7 = 0x40) of this register indicates whether or not the mouse button is being pressed. Note that this register is active low, meaning that if bit 7 is set, then the mouse button is not being pressed, while if it is cleared, the button is being pressed.

```
/* mouse.c */
#define Register_Address 0xbfe001;
char *CIAA_Reg1;          /* access to CIAA register 1 */
main()
{
   CIAA_Reg1 = (char *)Register_Address;
   printf (" <mouse button> running... Waiting for left
                                        mouse button !\n");
   while ((*CIAA_Reg1 & 0x40) == 0x40);
   printf ("Button pressed !!!\n");
}
```

2.5.2 Dynamic arrays

The second programming example concerns the creation of dynamic arrays. Let's say you want to store an arbitrary number of values. The number of values to be stored is not determined until the program is run (by the user, for example).

Normally the user is given some restrictions on the use of the program. Perhaps the maximum number of elements in a given array is limited to 100. This is not particularly elegant, however. It would be better if the maximum number of elements were limited only by available memory. Here again pointers can help us.

First define a pointer to the elements which make up the array:

```
int *array;
```

Then check to see how much memory is available with `AvailMem()`. It depends on the application whether you want this memory in chip memory (lower 512K) or fast memory.

```
size = AvailMem(MEMF_CHIP);
```

Now calculate how many array elements can be stored in this space:

```
NumElements = size/sizeof(*array);
```

You can now ask the user how many elements are used (elements). Then you just have to allocate the memory:

```
array = AllocMem(sizeof(*array) * elements, MEMF_CHIP);
```

Now you can access the individual array elements with `array[x]` and they are stored in the allocated memory.

2.5.3 Function tables

Function tables are arrays which store the addresses of individual routines. Such an array is defined as follows:

```
void (*functions[4])();
```

This is an array of pointers to functions with the return value `void`. After the definition, you just have to assign the address of the individual functions to the array elements:

```
/*******************************************************/
/*                      Table.c                        */
/*                (c) Bruno Jennrich                   */
/*                                                     */
/* This program shows you how to                       */
/* easily program jump tables.                         */
/*                                                     */
/* The functions are stored in an array                */
/* which can be accessed with an index variable.       */
/*******************************************************/

void (*functions[4])(); /* Functions is a four-element */
                        /* array of pointers to functions */


/* The functions */


void Funct0()
{
   printf ("Function 0 !\n");
}

void Funct1()
{
   printf ("Function 1 !\n");
}

void Funct2()
{
   printf ("Function 2 !\n");
}

void Funct3()
{
   printf ("Function 3 !\n");
}
```

```
main(argc, argv)
int    argc; /* (argument count) */
char **argv; /* (argument values) */
{
   int i;

   functions[0] = Funct0;
   functions[1] = Funct1;
   functions[2] = Funct2;
   functions[3] = Funct3;

  if (argc != 2)
   {
    printf (" Call: < TABLE Number_of_the_function >\n");
         exit(0);
      }

   i = atoi (argv[1]);

  if ((i>=0) && (i<=3)) (*functions[i])();
  else printf ("Number_of_the_function must be between 0\
                                        and 3\n");
}
```

3. Intuition and C

3. Intuition and C

This chapter concerns programming the Amiga operating system, especially system programming using Intuition. What do we mean by system programming? Here are some examples:

- How do I tell my Amiga operating system that I want to use a certain graphic mode?
- How can I tell the system that the user has made some inputs (with the mouse or with the keyboard)?

These are two questions which come under the topic "system programming." There are certainly others, but this chapter answers these questions precisely.

Intuition is a type of communication (user interface) between you, the user and the Amiga. By communication we mean any realtime exchange of information. The keyboard and mouse are the most obvious input devices—both are included with the computer, but there are certainly other devices. Output occurs mainly to the video screen. There are also printers and disks, but Intuition has no control over these devices.

Screens

What does Intuition offer? Let's look at what Intuition can do with output. When booting with the Workbench disk you immediately see two output displays offered by Intuition. The first and most important is the screen, which forms the foundation of every output. An important example is the Workbench screen, which appears when booting. It accompanies the user through everything he does, and acts as the basic output device for many programs.

Windows

Output rarely appears only on the screens, since they're difficult to implement. To make things easier, you can use windows, the second output level of Intuition.

Gadgets

Now we come to user input. Gadgets are the simplest input media. They are found in both screens and windows. The simplest gadgets react to a single mouse click. Naturally there are more complex types into which you can enter text, but these involve keyboard access. First we'll concentrate on mouse input, then look at keyboard input.

Menus were intended for elementary program-specific functions. They are also controlled by the mouse, but the emphasis is on text selection, while the gadgets are generally represented with graphics. There are, however, exceptions to these rules.

3.1 Windows

As we mentioned in the introduction to this chapter, windows are the most important input/output medium.

We wanted to present a detailed description of a complete application you could program in C, then continue to improve upon it throughout this book. We decided on creating a text editor for C programs. Therefore this section supplies you with information about programming fundamental I/O, and selecting what to use from the many options. You will also see how much Intuition offers to the programmer.

After listing the various parameters available, we'll see how to configure one of the windows managed by your program. As a result you will learn how to make use of the various Intuition commands.

In addition, we'll write a general window routine which you can use in your own programs with only a few changes. You should prepare a disk for this data, since you'll be using the modules created throughout the course of this book.

3.1.1 Window parameters and selecting them

You need a number of parameters for creating a window. These parameters can control every conceivable aspect of the window. To lead into this, remember the familiar properties found in the average Workbench window:

External features

Sizes and positions of every window can be adjusted. You can set the following values: X position (upper left corner), Y position (upper left corner), width and height.

If you want to change the size of a window, it is desirable to set maximum and minimum window sizes. When working with the Workbench you can generally resize the directory windows as much as you want. However, requesters displayed on the WorkBench screen can only be made smaller. As the programmer, you can set the maximum and minimum sizes with `MaxWidth`, `MaxHeight`, `MinWidth` and `MinHeight`.

In addition, you can set the colors used to draw the window border and the gadgets found within the window, based upon the screen colors. The

two colors are designated `DetailPen` and `BlockPen`. Each window also has a title line that contains text (again, programmer controlled).

For those not satisfied with the system gadgets offered by and in part managed by Intuition, you can add more gadgets to a window. This is how the scroll bars in the directory windows are managed, for example, because they are not defined as system gadgets. As a programmer you can construct any gadget you can think of.

The last external property you can specify about a window you have created is that system gadgets can be included or omitted, depending on what makes sense for the given application.

Internal features

Now that we've discussed the external parameters of windows, let's look at those which you might not recognize by just looking at a window.

One seldom used option is to redefine the appearance of the checkmark used to check menu items. You've probably heard of programming Intuition to manage a custom screen for you. Let's assume that you have a custom screen and you want to put your new window in it. A flag tells the operating system that the window is to appear on your screen and not on the Workbench screen. In addition, Intuition needs to know on which custom screen the window should appear, since it is possible to have more than one custom screen open at a time. Therefore, if you want to open a window on a custom screen, designate this with a flag and store the pointer to your screen in the window structure.

Program-internal features

The range of applications for which windows are intended is so broad that more than one type is needed. Therefore, various methods have been prepared from which programmers can choose one or more types.

Refreshing window displays

Window contents, which may include text as well as graphics, should be preserved independent of other windows. This should even be the case when other windows overlay it and cover parts of it. If Intuition were to simply store the hidden areas temporarily (a process called *buffering*), it would involve a great deal of memory. In some cases buffering may not be necessary. If you know what is in the window and it is relatively easy to reconstruct, then you can do all this in the program. The best known examples are the directory windows in the Workbench. There the system simply redraws the icons after other windows cover them. Workbench restores the graphics without buffering from Intuition. This takes up much less memory when multiple windows overlap each other.

You must consider the uses for a window when developiong a program. There are three basic application needs:

1. Programs where the contents of the windows is completely known and whose data are stored in a different form. An example of this is

word processing, where the text appears as graphic information on the screen, and is also stored in the text buffer. For such applications it's best to perform the reconstruction yourself. This takes as little memory as possible away from other programs running at the same time.

2. Programs unable to reconstruct a damaged portion of a window in a reasonable amount of time. Here Intuition handles the storage of the overlapped areas to save work on the program's part. This problem often occurs with graphic programs, where the time spent redrawing the image far outweighs the time involved in buffering it. You must strike a careful balance between memory usage and time. Memory usage should be carefully planned on any computer, even in a computer with 1 meg of RAM.

3. Programs which require more graphic display than is actually displayable. For this category a completely new method has been developed. The program displays its graphic in a completely independent bit-map, and Intuition uses the desired portion from this bit-map for the window. Therefore the graphic memory supplies reference points for reconstructing the window.

Let's summarize the three modes in a table. The table contains the names of the flags which must be set for the corresponding mode. Note that each mode is mutually exclusive of the others.

Table 3.1: Window refresh

Refresh	Flag	Memory usage
Simple	SIMPLE_REFRESH	None, since Intuition doesn't bother with the window
Supported	SMART_REFRESH	Corresponds to hidden areas
Complete	SUPER_BITMAP	The entire graphic in the window is buffered

Borderless windows

Your window doesn't always have to appear with the familiar border, title line, gadgets and border lines.

Viewed at its lowest level, a window is just an area on the screen that separates its contents from other areas present. The border which Intuition draws merely acts as a visible indicator of the border. If all the windows on the Workbench were drawn without this border, it would be very confusing to the user.

However, borderless windows can serve a practical purpose in many applications. You can prevent the border from being drawn when setting up a new window by setting the BORDERLESS flag. Just be sure that a borderless window won't confuse the user.

BACKDROP windows

The title of this paragraph refers to a window which no longer has all the properties of a normal window. It will always be the window farthest back, and can be overlapped by any other windows. The user cannot bring this window to the foreground. This window has no front or back gadgets—only the close gadget is allowed.

You can use this BACKDROP window type as the basis for any application. This BACKDROP window can hold the background for an animation program, or can display program status information, or even hold gadgets for accessing program tools. The BACKDROP property supports these applications, since a new window cannot be placed behind it. Intuition manages the BACKDROP window, so further programming is unnecessary. If you want to use this window type, you must set the BACKDROP flag.

An interesting combination would be a BORDERLESS-BACKDROP window, which would be very useful for the examples just mentioned. But other types can be combined as well.

An extra window border

Earlier we described how to create a window which had no border. Here we describe the opposite situation. You know that a normal window is just a rectangular area with a visual border. A few problems result from this: If you write a drawing program which uses a normal window, you can easily overwrite the title bar, the gadgets or the border. The window only prevents you from drawing outside of the window's border—the rest of the window is open game.

GIMMEZERO ZERO

How do you cure this problem? The simplest way; avoid drawing outside the window's inner surface. But let's be realistic, users don't blindly accept the universal programmer's instruction, "Don't do it." The developers of Intuition created GIMMEZEROZERO mode to solve this problem. This mode changes some window settings. The window consists of two fields: One field contains all of the gadgets, the title bar and the border. The other field is the surface in which the user can draw without crossing over into gadgets or border art. The name GIMMEZEROZERO comes from the fact that this inner window always has the upper left corner coordinates 0,0, regardless of the size and location of the window.

This mode requires more memory and processing time. Also, the portion in which the drawing program can operate has different dimensions from those specified in the window definition. This window type needs two additional variables which you read to find out the size of the actual drawing surface. These variables are called `GZZHeight` and `GZZWidth`. This area lies within the following values: `BorderLeft`, `BorderTop`, `BorderRight` and `BorderLeft`.

Since the mouse coordinates of a window always refer to the entire window area, a GIMMEZEROZERO window can cause certain problems. Intuition also has relative mouse coordinates to the GIMMEZEROZERO window: `GZZMouseX` and `GZZMouseY`.

Scrolling data

Intuition offers the programmer the ability to manage a much larger graphic surface than can be displayed on the screen. `SUPER_BITMAP` mode displays a window type where you can display part of a bit-map of almost any size in a window. If you want to see a different section, the system displays that section. This is useful when scrolling graphics.

If you want to use this window type, set the `SUPER_BITMAP` flag in a program. It's possible to combine this with `GIMMEZEROZERO` or `BACKDROP`.

System gadgets

Gadgets are fields which can be activated by moving the mouse pointer onto the gadget and clicking the left mouse button. Intuition offers four gadgets as a default set. These four system gadgets support functions which cannot normally be handled by the program. System gadgets also have a variety of properties. Their positions and appearance are preset; only the color can be changed. They can always be recognized by the user in any program, and the user doesn't have to guess what they mean. Let's look at each of them.

Close gadget

The close gadget closes a window. You are familiar with the procedure from the Workbench: You click on the gadget in the upper left corner of the window to close the window. The close gadget appears as a box with a large point in its center.

Drag bar

The drag bar appears as three heavy lines in the title bar of a window. You can use it to change the position of the window. Move the mouse pointer onto the drag bar and press and hold the left mouse button. As long as you hold down the mouse button the window moves with the mouse pointer.

Front and back gadgets

These gadgets always come as a pair. They change the priority of overlapping windows. The front and back gadgets are found in the upper right corner. The light rectangular surface is the important part: It represents the current window's new position.

Sizing gadget

If a program allows you to change the size of a window, the window has a sizing gadget in its lower right corner. It depicts two different size rectangles. Click on it and press and hold the left mouse button. Now moving the mouse changes the window's size. The new size can only be set within the predefined minimum and maximum sizes.

Since the sizing gadget enlarges the border of the window, the area in which the Intuition gadgets are located, the programmer must also specify where this border is taken from the actual output window. The right border or lower border are possibilities, depending on whether you want to have more rows or more columns in the window. It is also possible to reserve both areas for Intuition.

It should be noted that Intuition receives a message from the close gadget but that Intuition doesn't close that window. The reaction to the close gadget depends on the program controlling the window. Just as the program receives a message when the user wants to close the window, it also receives a message when the size of the window has been changed, since this must be taken into account by the program.

3.1.2 How Intuition manages windows

You now know about the basic window parameters. Before you try out these parameters, you should know the principles used to access Intuition commands. You might also be interested in internal data transmission in the Amiga.

3.1.2.1 Accessing the Intuition library

The entire Amiga operating system is made up of system routines called *libraries*. A library consists of various routines, which accept parameters and sometimes return values.

Not all of the libraries are available when the user turns on the computer. If they are already loaded, a program must still indicate that it wants to use a certain library, since access requires the use of a pointer. This is an EXEC command (EXEC is the library which is active immediately after powering up) which, like all EXEC commands, is available to every program:

```
library = OpenLibrary(LibraryName, Version)
             D0          A1          D0
```

You pass the library's function (in this case `intuition.library`) and version number. Newer library versions may contain functions not found in older libraries. You can prevent an older version of a library from being loaded by specifying the version number.

We will use the basic commands, however, since there are already windows in Intuition. The first lines look like this:

```
/* OpenIntuition.c, section 3.1.2.1.A */
struct IntuitionBase *IntuitionBase;
void                 *OpenLibrary();

if (!(IntuitionBase = (struct IntuitionBase *)
    OpenLibrary("intuition.library", 0L)))
```

```
{
printf ("Intuition library not found!\n");
exit (FALSE);
}
```

How it works The first structure definition introduces a pointer which points to the Intuition function list. The compiler requires this pointer since all Intuition access occurs relative to this pointer.

An additional untyped pointer also appears for the `OpenLibrary` function. You may want to open more than the Intuition library, in which case type conversion may cause problems.

The `if` test first calls the function itself. Then its return value (the pointer to the Intuition functions) is tested for validity. This value must be something other than null, in which case it represents the memory address at which the library begins. A null value indicates an error. If the `if` condition is true, the program displays a message in the AmigaDOS window and exits.

Now you need to know how to conclude library access. When you actually obtain access (and only then) you can simply close the library again with the EXEC function `CloseLibrary()`:

```
CloseLibrary(LibraryPointer)
     -414          A1
```

In your application this would appear as:

```
CloseLibrary(IntuitionBase);
```

3.1.2.2 The `NewWindow` structure

Now you know how to open and close a library. Let's look at windows again. All of the properties discussed at the beginning of this chapter can be specified when you open or change a window later. A structure is used to initialize the values for Intuition.

The `NewWindow` structure listed below contains all of the values required by Intuition to configure a new window. The structure looks like this:

```
/* NewWindow structure */
struct NewWindow
{
0x00  00  SHORT LeftEdge;
0x02  02  SHORT TopEdge;
0x04  04  SHORT Width;
0x06  06  SHORT Height;
```

```
0x08  08  UBYTE DetailPen;
0x09  09  UBYTE BlockPen;
0x0a  10  ULONG IDCMPFlags;
0x0e  14  ULONG Flags;
0x12  18  struct Gadget *FirstGadget;
0x16  22  struct Image *CheckMark;
0x1a  26  UBYTE *Title;
0x1e  30  struct Screen *Screen;
0x22  34  struct BitMap *BitMap;
0x26  38  SHORT MinWidth;
0x28  40  SHORT MinHeight;
0x2a  42  USHORT MaxWidth;
0x2c  44  USHORT MaxHeight;
0x2e  46  USHORT Type;
0x30  48
}
```

As you know, a structure consists of a many different variables. You may wonder why numbers precede each variable name in the variable declaration. These numbers indicate the offset of each variable from the start of the structure. This can be helpful when working with the debugger, or if you are using assembly language. The numbers do not coincide with the actual C structure declaration.

The first column contains the offsets in hexadecimal notation, and the second in decimal notation. The last number indicates the total length of the structure.

All library communication occurs in predefined functions to which you pass variables (e.g., `OpenLibrary` and `CloseLibrary`). Intuition, like most other libraries, uses structures of its own for various management purposes.

Creating a `NewWindow` structure

Here we want to explain the setup of the `NewWindow` structure. In addition, we'll show you two functions which will help you in writing your own programs later.

The first function comprises everything which must be opened, initialized, allocated or organized at the start of the program. This includes memory allocation, opening libraries, clearing fields, etc.

The second function comprises everything which must be closed at the end of the program. If you don't close a library after you finish with it, the memory allocated for managing it won't be returned to the system.

The solution to these programs looks quite simple, one small problem needs a solution: Something may go wrong during initialization. In this case you should display an error message and terminate the program. You cannot forget to close everything which has already been opened, however. And here is where the problem occurs.

If you simply jump to the close routine, it might try to close things which weren't open in the first place. This guarantees a system crash. So, let's write a close function which examines the pointers obtained during initialization. This function should only close open items, and leave everything else alone. Let's look at the two functions:

```
/*************************************/
/*                                   */
/* 3.1.2.2.B   Open_All.c            */
/*                                   */
/* Function: Open everything we need */
/* ================================= */
/*                                   */
/* Author:  Date:      Comment:      */
/* ------  ----------  ----------    */
/* Wgb     16.10.1987  Intuition only */
/*                                   */
/*************************************/

Open_All()
   {
   void          *OpenLibrary();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition library not found!\n");
      Close_All();
      exit(FALSE);
      }
   }

/*************************************/
/*                                   */
/*  3.1.2.2.C   Close_All.c          */
/*                                   */
/* Function: Close everything opened */
/* ================================= */
/*                                   */
/* Author:  Date:      Comment:      */
/* ------  ----------  ----------    */
/* Wgb     16.10.1987  Intuition only */
/*                                   */
/*************************************/

Close_All()

   {
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

How it works The first function should already look familiar to you. The contents of the second function are almost silly, but the important part is the `if` test. The program would crash without this test if the Intuition library could not be opened for some reason.

Both functions look quite simple at the moment, but the basic form will continually expand as we continue. Enter them carefully and save them both on your program disk and in a subdirectory created specifically for storing function blocks. Later, when either of them is required, we will not present the entire listing, just references to the existing part and the additions required.

Now you can start writing your first program to open and close a simple window. Let's specify the properties for your window:

Window size

The position and size of the window don't matter too much for your test. Just be sure that the width and height are large enough to fit gadgets and the window border. We set ours at position 160, 50 with the dimensions 320, 150. Write the values in a `NewWindow` variable called `FirstNewWindow`:

```
FirstNewWindow.LeftEdge    = 160;
FirstNewWindow.TopEdge     = 50;
FirstNewWindow.Width       = 320;
FirstNewWindow.Height      = 150;
```

Since you don't want to omit the sizing gadget, set the screen dimensions to the maximum. Remember, the minimum shouldn't be too small.

```
FirstNewWindow.LeftEdge    = 160;
FirstNewWindow.TopEdge     = 50;
FirstNewWindow.Width       = 320;
FirstNewWindow.Height      = 150;
```

Since you don't know how to manage screens on your own yet, use the Workbench screen. The window type is set and you won't have to find a pointer:

```
FirstNewWindow.Type         = WBENCHSCREEN;
FirstNewWindow.Screen       = NULL;
```

For now, we'll skip menus and custom gadgets:

```
FirstNewWindow.FirstGadget = NULL;
FirstNewWindow.CheckMark   = NULL;
```

Title bar

Now let's get to the graphic details. First you have the window title. You can pass a pointer to a string ending with a null, or you can pass a null pointer to indicate an untitled window. If no system gadget is permitted, then no title bar is displayed.

When using the Aztec compiler a type `cast` should precede the string to avoid a warning from the compiler.

```
FirstNewWindow.Title   = (UBYTE *)"System programming test";
```

Two interesting and seldom used parameters are the two pen colors. Depending on the screen, they can accept values from 0 to 31, which naturally depends on the number of bit-planes. A special value is -1 (0xff), which uses the default screen color values. This assigns all of the windows in the screen the same colors, allowing global color changes as well. The example uses Workbench color values, but not the default values. This makes the new window independent of the superordinate screen's parameters, even if they currently contain the same values.

```
FirstNewWindow.DetailPen  = 0;
FirstNewWindow.BlockPen   = 1;
```

You can only initialize two more parameters. These two flag parameters present so many options that we must lay out descriptions of each flag. They can be divided into six basic groups: Gadget flags, sizing gadget position, refresh flags, window type flags, mouse flags and miscellaneous flags.

Gadget flags

These flags must be set according to the system gadgets you want in the window.

`WINDOWDRAG` Adds a drag bar to the title bar.

`WINDOWDEPTH` Adds front and back gadgets.

`WINDOWCLOSE` Adds a close gadget readable by the program.

`WINDOWSIZING` Adds a sizing gadget for window size control.

Sizing gadget position

If the sizing gadget is used, two additional flags must be integrated, because all system gadgets are placed in the window borders. It should be clear that the first three are placed in the upper border. You have two options for the sizing gadgets:

`SIZEBRIGHT` The sizing gadget goes in the right border.

`SIZEBOTTOM` The sizing gadget goes in the bottom border.

Refresh flags

Window contents can easily be destroyed if the window is overlapped by another window. Intuition offers various ways of getting around this problem:

`SIMPLE_REFRESH`
This refresh status means that Intuition does not do anything to restore the window. Any reconstruction must be handled by the program itself.

SMART_REFRESH
: Here Intuition buffers every hidden area of the window and then restores the area when made visible. This causes problems only when resizing windows. If you make the window smaller, the eliminated portions are not buffered and not restored.

SUPER_BITMAP
: This type of refresh uses the most memory. In some cases you may need to store a large graphic and display it in a window. This is both a window type and a refresh mode, in that the window can always be restored by the information contained in the bit-map, and it also manages graphics larger than the display window.

NOCAREREFRESH
: The SMART_REFRESH and SIMPLE_REFRESH window types require at least some program cooperation. Intuition loses its way when the window is resized with SMART_REFRESH and for SIMPLE_REFRESH the overlaid area must be redrawn. The program receives a message saying that something must be done. The message port and how it is used are explained later. If you don't want to do anything, then there is no need to send these messages. This flag tells Intuition that you don't want this information.

Window type flags

We have already discussed various window types. Here are the flag names with brief comments:

BACKDROP
: This window always has lower priority than other windows. Only one of these windows should exist per screen.

SUPER_BITMAP
: Although SUPER_BITMAP is also a refresh type, it also applies as a window type. There is only a flag.

BORDERLESS
: Intuition does not draw a window border.

GIMMEZEROZERO
: Intuition keeps the border and all gadgets separate from the window's contents.

Mouse flags

These two flags control the status of the mouse.

REMOUSE
: If you set this flag you get continuous messages about the mouse's position.

RMBTRAP — All mouse signals from the right mouse button convert into signals from the left button. This can be used when there are no menus present.

Miscellaneous This last flag determines whether or not the window is active when open.

ACTIVATE — If this flag is set, the window is active immediately when it is opened. This is often important since input is always directed to the active window. This can cause problems if the user has already entered something and this flag opens a new window. Use such windows with caution!

All of these flags are represented as set or cleared bits in the flag entry of the NewWindow structure. When programming in C you can use the symbolic names. But for assembly language programmers and those who want more information, you need the hex values for the flags. Here's a table of the flags and their values:

Table 3.2: Window flags

Flag Name	Hex Value	Gadget group
WINDOWSIZING	0x00000001L	System gadgets
WINDOWDRAG	0x00000002L	
WINDOWDEPTH	0x00000004L	
WINDOWCLOSE	0x00000008L	
SIZEBRIGHT	0x00000010L	Gadget position
SIZEBBOTTOM	0x00000020L	
NOCAREREFRESH	0x00020000L	Refresh types
SIMPLE_REFRESH	0x00000040L	
SMART_REFRESH	0x00000000L	
SUPER_BITMAP	0x00000080L	
BACKDROP	0x00000100L	Window types
GIMMEZEROZERO	0x00000400L	
BORDERLESS	0x00000800L	
REPORTMOUSE	0x00000200L	Mouse flags
RMBTRAP	0x00010000L	
ACTIVATE	0x00001000L	Miscellaneous

This is the definition you will use in your NewWindow structure:

```
FirstNewWindow.Flags    = WINDOWDEPTH | WINDOWSIZING |
                          WINDOWDRAG | WINDOWCLOSE |
SMART_REFRESH;
```

Now that you have some knowledge of the simple flags, we'll spend a little time looking at the IDCMP (Intuition Direct Communication Message Port) flags. First we need a definition of the IDCMP.

The IDCMP is a very complex data channel which carries information about all of the areas managed by Intuition. Here we'll just list them, since the topic is so comprehensive that we dedicated an entire section to it (Section 3.6). We include some information about it here because any interaction with Intuition requires the IDCMP.

You can get information about the following Intuition peripherals: Windows, gadgets, menus, mouse, keyboard, disk, Preferences, clock.

But since you don't know anything about this yet, we'll use a little trick. Instead of using a close gadget to close the window opened by the program, we'll close it after a short time delay. This way you can bypass using the IDCMP for now:

```
FirstNewWindow.IDCMPFlags = NULL;
```

This concludes the structure definition of FirstNewWindow. Take a look at the program below. It consists of the two functions Open_All() and Close_All(), the window structure, and a main function:

Program 3.1: First window

```
/*************************************/
/*                                   */
/*  3.1.2.2.D  First_window.c        */
/*                                   */
/* Program: Window local definition  */
/* ================================= */
/*                                   */
/* Author:  DAte:      Comment:      */
/* ------  ----------  ----------    */
/* Wgb     10/16/1987  first test    */
/*                     window        */
/*                                   */
/*************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;


struct NewWindow FirstNewWindow;


main()
   {

   FirstNewWindow.LeftEdge    = 160;
   FirstNewWindow.TopEdge     = 50;
```

```
    FirstNewWindow.Width      = 320;
    FirstNewWindow.Height     = 150;
    FirstNewWindow.DetailPen  = 0;
    FirstNewWindow.BlockPen   = 1;
    FirstNewWindow.IDCMPFlags = NULL;
    FirstNewWindow.Flags      = WINDOWDEPTH | WINDOWSIZING
                              | WINDOWDRAG | WINDOWCLOSE
                              | SMART_REFRESH;
    FirstNewWindow.FirstGadget= NULL;
    FirstNewWindow.CheckMark  = NULL;
    FirstNewWindow.Title = (UBYTE *)"System test";
    FirstNewWindow.Screen     = NULL;
    FirstNewWindow.BitMap     = NULL;
    FirstNewWindow.MinWidth   = 100;
    FirstNewWindow.MinHeight  = 50;
    FirstNewWindow.MaxWidth   = 640;
    FirstNewWindow.MaxHeight  = 200;
    FirstNewWindow.Type       = WBENCHSCREEN;

    Open_All();

    Delay(180L);

    Close_All();
    }


/*************************************
 *                                   *
 * Function: Open library and window *
 * ================================= *
 *                                   *
 * Author:  Date:      Comment:      *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987                *
 *                                   *
 *                                   *
 *************************************/

Open_All()

    {
    void          *OpenLibrary();
    struct Window *OpenWindow();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
       {
       printf("Intuition library not found!\n");
       Close_All();
       exit(FALSE);
       }

    if (!(FirstWindow = (struct Window *)
        OpenWindow(&FirstNewWindow)))
       {
```

```
        printf("Window will not open!\n");
        Close_All();
        exit(FALSE);
        }
    }

/*************************************
 *                                   *
 * Function: Close everything opened *
 * ================================= *
 *                                   *
 * Author:  Date:      Comment:      *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  just Intuition *
 *                     and window    *
 *                                   *
 *************************************/

Close_All()

    {
    if (FirstWindow)      CloseWindow(FirstWindow);
    if (IntuitionBase)    CloseLibrary(IntuitionBase);
    }
```

How it works The listing contains a few things that may be unfamiliar to you. First, the `Open_All()` function has been modified to open the window, and `Close_All()` has been correspondingly modified to close the window.

How does a window open once you define a `NewWindow` structure? It is very important that the structure be defined before the `main()` function, otherwise the structure values are not available to the other functions. The `OpenWindow` simply passes the address of the definition structure (`NewWindow`). As the return value you get a pointer to the window structure which was created based on your data. It also contains additional management information. `OpenLibrary` must be checked to ensure there were no errors (you can't assume that everything worked properly with `OpenWindow()`). There are many things that could cause an error in `OpenWindow()`, such as running out of memory. The `NewWindow` structure can then be erased again. It is only needed for initialization.

```
WindowPointer = OpenWindow(NewWindowStructure)
      D0           -204           A0
```

Closing the window has also been added to the `Close_All()` function. The structure is similar to what you have seen already. The important thing is that the window closes first, and then the Intuition library. If these actions were reversed, the operating system would no longer be able to find the `CloseWindow()` function and the program would crash.

```
CloseWindow(WindowPointer)
    -72          A0
```

A time delay appears between `Open_All()` and `Close_All()` in the main program with the `Delay()` function. This operating system puts the task of your program into wait status until the specified time has elapsed (time is specified in "ticks", 60 ticks per second), since this program doesn't have the ability to read the close gadget in order to close the window again. Instead of the `Delay()` function you could also use a `for` loop to achieve the delay. But this would have slowed down the multitasking even though you were only waiting.

Let's look at the start of the program and its comment blocks. Before the `main()` function you define two structures which you need for your window. The first is the `NewWindow` structure under the name `FirstNewWindow`. This allocates space for the information which Intuition requires to set up your window. The second structure is a pointer to a `Window` structure. A `Window` structure contains much more information than a `NewWindow` structure, and is linked to other windows in the screen. The construction of the system which Intuition sets up and manages looks like this:

"Ground zero" is a screen, generally the Workbench screen. This `Screen` structure, explained in the next section, contains a pointer to the first window in the screen. The `Window` structure itself has a pointer to the following windows. If there are several screens, then the first screen also stores a pointer to the next screen, which can contain the same window chaining. Here is a figure to clarify the structure·

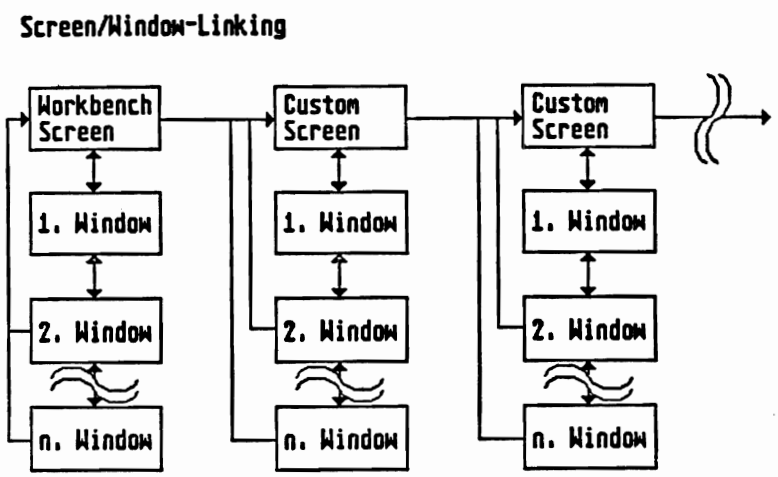


Figure 3.1

The structure definition described above must have access to the operating system's Intuition functions in order for our new functions to work. These can only be accessed once the library has been opened. You must know the address of the library, stored in the pointer `*IntuitionBase`.

Finally, we should say something about comments. Comments always contain information about the author and the creation date. This is the

date the first lines were written. A new date represents a fundamental change in the routine. The purpose of the program or function is also explained in the header.

3.1.2.3 The `Window` structure

The `Window` structure stores all of the information needed for Intuition to manage a window. Here it is:

```
struct Window
{
0x00   00  struct Window *NextWindow;
0x04   04  SHORT LeftEdge;
0x06   06  SHORT TopEdge;
0x08   08  SHORT Width;
0x0a   10  SHORT Height;
0x0c   12  SHORT MouseY;
0x0e   14  SHORT MouseX;
0x10   16  SHORT MinWidth;
0x12   18  SHORT MinHeight;
0x14   20  USHORT MaxWidth;
0x16   22  USHORT MaXHeight;
0x18   24  ULONG Flags;
0x1c   28  struct Menu *MenuStrip;
0x20   32  UBYTE *Title;
0x24   36  struct Requester *FirstRequest;
0x28   40  struct Requester *DMRequest;
0x2C   44  SHORT ReqCount;
0x2E   46  struct Screen *WScreen;
0x32   50  struct RastPort *RPort;
0x36   54  BYTE BorderLeft;
0x37   55  BYTE BorderTop;
0x38   56  BYTE BorderRight;
0x39   57  BYTE BorderBottom;
0x3A   58  struct RastPort *BorderRPort;
0x3E   62  struct Gadget *FirstGadget;
0x42   66  struct Window *Parent
0x46   70  struct Window *Descendant;
0x4A   74  USHORT *Pointer;
0x4E   78  BYTE PtrHeight;
0x4F   79  BYTE PtrWidth;
0x50   80  BYTE XOffset;
0x51   81  BYTE YOffset;
0x52   82  ULONG IDCMPFlags;
0x56   86  struct MsgPort *UserPort;
0x5A   90  struct MsgPort *WindowPort;
0x5E   94  struct IntuiMessage *MessageKey;
0x62   98  UBYTE DetailPen;
0x63   99  UBYTE BlockPen;
0x64  100  struct Image *CheckMark;
0x68  104  UBYTE *ScreenTitle;
```

```
0x6C 108  SHORT GZZMouseX;
0x6E 110  SHORT GZZMouseY;
0x70 112  SHORT GZZWidth;
0x72 114  SHORT GZZHeight;
0x74 116  UBYTE *ExtData;
0x78 120  BYTE *UserData;
0x7C 124  struct Layer *WLayer;
0x80 128  struct TextFont *IFont;
0x84 132
};
```

As with the NewWindow structure, the numbers preceding the structure element names are the offsets from the start of the structure.

Let's go through the parameters one by one.

*NextWindow — A pointer to the next Window structure in the screen. This pointer maintains the window chaining mentioned.

LeftEdge, TopEdge — Position of the window on the screen, as defined in NewWindow.

Width, Height — Dimensions of the window, as defined in NewWindow.

MouseX, MouseY — Mouse coordinates, relative to the upper left corner of the window.

MinWidth, MinHeight — Minimum values as defined in NewWindow.

MaxWidth, MaxHeight — Maximum values as defined in NewWindow.

Flags — Flag list, as defined in NewWindow.

*MenuStrip — Pointer to the window menu structure (see Section 3.8 for more information). No menus can be set through NewWindow. This must be done with a separate function.

*Title — Pointer to string containing the window title text.

*FirstRequester — Pointer to the first requester set in this window (see Section 3.5).

*DMRequest — Pointer to double menu requester (see Section 3.5.4).

`ReqCount` Counter for the requesters opened in this window.

`*WScreen` Pointer to screen in which this window opened.

`*RPort` Pointer to the `RastPort` setup for this window.

`BorderLeft, BorderTop, BorderRight, BorderBottom`
Border size specifications.

`*BorderRPort` Pointer to the `RastPort` for the border (used for `GIMMEZEROZERO` windows).

`*Firstgadget` Pointer to the first gadget in a linked list of all the gadgets in this window.

`*Parent, *Descendent`
Chaining previous and next window, to make opening and closing easier.

`*Pointer` Pointer to graphic for this window's mouse pointer (mouse pointer can be redefined for each window).

`PtrHeight, PtrWidth`
Mouse pointer size (X value cannot exceed 16).

`XOffset, YOffset`
Offsets specifying point in the pointer representing the click point.

`IDCMPFlags` The flags as defined in `NewWindow`.

`*UserPort` Message port for data communication.

`*WindowPort` Message port for data communication.

`*MessageKey` Message port for Intuition reports.

`DetailPen, BlockPen`
Pen colors as defined in `NewWindow`.

`*CheckMark` Pointer to the checkmark defined for the menus.

`*ScreenTitle` Pointer to the string for the screen title (can only be set by the function `SetWindowTitles()`). See explanation below.

`GZZMouseX, GZZMouseY`
Mouse coordinates for a `GIMMEZEROZERO` window (border is automatically taken into account and subtracted from the normal value).

`GZZWidth, GZZHeight`
: Window size for a `GIMMEZEROZERO` window.

`*ExtData`
: Pointer to external data structures (not used).

`*UserData`
: Pointer to a data structure set by the programmer.

`*WLayer`
: Pointer to the `Layers` structure of this window (`RPort->Layer` also contains the same value).

`*IFont`
: Pointer to the font used for Intuition text output in this window.

The `Window` structure contains some pointers to very important Intuition and graphic elements. The following figure should clarify the structure:

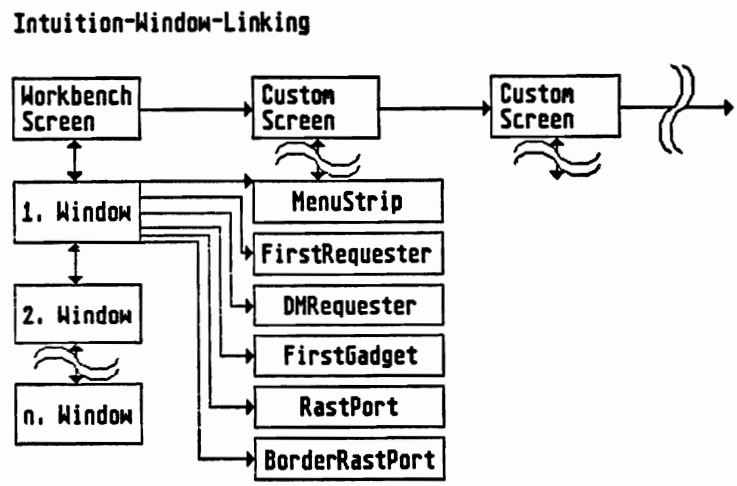


Figure 3.2

All of these values allow easy access. If you want to know the width of your `GIMMEZEROZERO` window, for example, you simply enter the following in your source code:

```
width = FirstWindow->GZZWidth;
```

You can also change a given value. Perhaps you want to use a different graphic for the checkmark which appears next to selected menu items. Just enter the following in your source code:

```
FirstWindow->CheckMark = NewCheckMark;
```

Later you'll see the importance of access to window parameters.

Let's return to the window program. You can see how simple programming in Intuition can be. That won't change in general, but typing in all that initialization is still a lot of effort. You have to find a

method to make structure definition easier. We wrote the following definition:

```
/* 3.1.2.3.B.1.firstnew_struct */
struct NewWindow FirstNewWindow =
   {
   160, 50,                  /* LeftEdge, TopEdge    */
   320, 150,                 /* Width, Height        */
   0, 1,                     /* DetailPen, BlockPen */
   NULL,                     /* IDCMP Flags          */
   WINDOWDEPTH |             /* Flags                */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                     /* First Gadget         */
   NULL,                     /* CheckMark            */
   (UBYTE *)"System test",
   NULL,                     /* Screen               */
   NULL,                     /* BitMap               */
   100, 50,                  /* Min Width, Height    */
   640, 200,                 /* Max Width, Height    */
   WBENCHSCREEN,             /* Type                 */
   };
```

This can be shortened to the bare essentials of information:

```
/* 3.1.2.3.B.2.short_firstnew */
struct NewWindow FirstNewWindow =
   {
   160, 50, 320, 150, 0, 1, NULL,
   WINDOWDEPTH|WINDOWSIZING|WINDOWDRAG|WINDOWCLOSE|SMART_REFRESH,
   NULL, NULL, (UBYTE *)"System  test",
   NULL, NULL, 100, 50, 640, 200, WBENCHSCREEN
   };
```

The advantage is clear: you have much less to type in because the structure name appears only once. It's up to you whether or not you want to include the comments.

The structure definition must precede the `main()` function. This makes it a global variable available to all functions. If you want to change a few values you can do so by accessing each element through its name. Note that when initializing structures in this manner the order dictates what you are actually initializing. Changing things around can easily lead to a crash.

3.1.2.4 A summary of window functions

Now that we are done with initialization, we can begin to build on what we have accomplished. You should use the `first_window.c` program as the basis for the demonstrations which follow. Keep the same window parameters you used there. It is easier if you use a global structure definition as explained above.

You have already seen the first two Intuition functions related to windows. For opening we used the `OpenWindow()` function and for closing we used `CloseWindow()`.

In the programs which follow, make sure that all windows which are opened are closed again. If a program ends without closing all of the windows it opened, the pointers to the remaining windows are lost and they remain lost until the computer is turned off or reset. A screen containing such windows cannot be closed either.

Changing window titles

Let's take a look at the rest of the window functions. The first is `SetWindowTitles()`. This lets you change the title of a window after it has been configured. To demonstrate this, insert the following lines into the `First_Window.c` program:

```
/* 3.1.2.4.A  Setwindowstitle.c /*
/* add after Delay(180L) */

SetWindowTitles(FirstWindow,
  "New window title", "The screen has a title now too!");
Delay(180L);
```

As you can probably tell from the text, the window is assigned a new title. Then the screen gets a title. When you click on a window the screen title line changes appearance. In most cases the program starts from the Workbench, and the screen doesn't have a special title. The same is true for the CLI. The text "Workbench Screen" is simply placed in the title line. You as the programmer can choose a different title for your active window. The format of this command looks like this:

```
SetWindowTitles(Window, WindowTitle, ScreenTitle);
    -276          A0        A1           A2
```

`Window` is the pointer to the structure of the window in question. The titles are passed as pointers to strings ending in null.

Now that you have seen how it works, you should give some thought to practical uses for `SetWindowTitles()`. Let's imagine a word processor. Here you probably work on several documents in succession or at the same time. It's vitally important that you know which document you're working on at the moment. You'd like the name of the

document to appear as the title of the word processor window. In addition, most programmers also add the path. This lets you see the name and path of the disk currently under access. The only way to give a title to the screen is to use SetWindowTitles().

If you want to change one title parameter of SetWindowTitles() without changing the other parameters (e.g., if you want to change the title of a window without changing the title of the screen), you can pass -1 as the parameter instead of a pointer. This leaves the appropriate title unchanged.

It is also possible to clear the title of either a window or the screen or both. This is done by passing null to a string instead of a pointer.

Window activation

The next window handling function first appeared in Version 1.2 of the Intuition library. The ActivateWindow() function allows the programmer to activate a window at any time during program execution. The only argument passed to it is a pointer to the Window structure. Insert the following lines in the original version of the FirstWindow.c program (not including the addition of SetWindowTitles() as entered above):

```
/* 3.1.2.4.B.activatewindow.c*/
/* place after Delay(180L) */

ActivateWindow(FirstWindow);
Delay(180L);
```

Compile the progam and then start it. Click in a different screen or in some other window as soon as the window appears, and watch what happens. Now your window is deactivated and after a delay it automatically reactivates. Here's the format for ActivateWindow:

```
ActivateWindow(Window);
     -450         A0
```

This function is important for input, which occurs only in the active window. If a program checks to see if a window is active which is not allowed to be active because no inputs are allowed, it can activate another window. This prevents the user from entering information in that window. See Section 3.7 (Reading the IDCMP flags) for a sample application.

Moving windows

It may also become necessary to change the position of a window. Perhaps a new window must be opened at the same position as the old one, but you want the old window to remain visible. You could leave this up to the user, but it might get aggravating if it happened often. If you as the programmer know when it is necessary to change the position of a window, then you can use the MoveWindow() function. The arguments you pass are the window pointer and two relative values which specify the movement in the X and Y directions.

But there is a danger associated with these delta (movement) values. If you move the window even one pixel outside the screen, the system crashes. Therefore you must check delta values. To do this we refer to the information in the `window` structure, which contains the current window position. Insert the following into the `FirstWindow.c` program:

```
/* 3.1.2.4.C. movewindow.c */
/* insert after line Delay(180L) */

DeltaX = FirstWindow->LeftEdge;
DeltaY = FirstWindow->TopEdge;

MoveWindow(FirstWindow, -1*DeltaX, -1*DeltaY);
Delay(180L);
```

Before you save the source code for this program remember to declare the two delta variables as `short`, to match the window structure. Access to the Amiga library requires 32 bit integers so be sure to use the +L option of the Aztec C compiler to compile this program so all integers are 32 bits long. When the new program is then started, the newly opened window moves to the upper left corner of the screen after a short pause.

Here is the format of the new function:

```
MoveWindow(MyWindow, DeltaX, DeltaY);
   -168        A0          D0       D1
```

Sizing windows

A function in a similar category is `SizeWindow()`. You may want to change the size or shape of a window from within the program, in cases when the user doesn't have the option of changing it himself. Good programs often give the user a gadget to click on to either reduce or enlarge the size of a window.

The first example appears below. A program at the end of this section contains the second example. Add these lines to `FirstWindow.c` program:

```
/* 3.1.2.4.D.sizewindow.c */
/*  insert after Delay(180L) */

DeltaX = FirstWindow->LeftEdge - FirstWindow->MinWidth;
DeltaY = FirstWindow->TopEdge  - FirstWindow->MinHeight;

SizeWindow(FirstWindow, -1*DeltaX, -1*DeltaY);
Delay(180L);
```

Remember to declare the variables as `short`. Be sure to use the `+L` option of the Aztec C compiler to compile this program so all integers are 32 bits long. This window type requires that you assign values to the `MinWidth` and `MinHeight` variables. Otherwise problems can arise when you reduce the window to the size of a single pixel. This

error can occur only if you have not selected a sizing gadget, and did not have to set the minimum and maximum sizes.

The format is very similar to that of the MoveWindow() function:

```
SizeWindow(MyWindow, DeltaX, DeltaY);
   -288       A0        D0       D1
```

Make sure you don't use any illegal values for the deltas. The window may not be made smaller or larger than the screen allows.

Limiting window size

WindowLimits() allows you to set the maximum and minimum sizes of a window after opening the window.

For example, if your program recognizes that it can no longer enlarge a graphic window due to memory limitations, you can simply update the maximum size values. The user can then only reduce the window's size. The example below, which can again be inserted into FirstWindow.c program, reads the old maximum values and then changes the old maximum values to new maximums, after the initial delay. The window can then be enlarged only up to the new limit.

```
/* 3.1.2.4.E.windowlimits.c */
/* insert after Delay(180L) */

MinWidth    = FirstWindow->MinWidth;
MinHeight   = FirstWindow->MinHeight;

NewMaxWidth = 200;
NewMaxHeight= 100;
Success = WindowLimits(FirstWindow, MinWidth, MinHeight,
                       NewMaxWidth, NewMaxHeight);
Delay(180L);
```

Before you can try this out, you must declare the limit values as USHORT and the result value (Success) as BOOL. Be sure to use the +L option of the Aztec C compiler to compile this program. The routine returns TRUE if values were all within the allowed limits.

If the maximum values are reset and the window is already larger than the new limits, the program returns FALSE as the result. Here is the format of the function call:

```
Result = WindowLimits(MyWindow, MinWidth, MinHeight,
  D0       -318          A0       D0         D1
                       MaxWidth, MaxHeight);
                          D2        D3
```

Moving windows to front and back

A newly opened window is always placed in front of all others. The fact that one window can cover all the others should be familiar to you from your daily use of Intuition. The programmer needs to know how to put a window into the background when it is not currently needed, and then returns it to the foreground. The user controls this procedure with the front and back gadgets. There are two corresponding Intuition functions: `WindowToBack()` and `WindowToFront()`.

Both functions require just the pointer to the `window` structure as the argument. Since you probably have the DOS window open when you start all of these programs, you can see how these functions work. Add the following lines to `FirstWindow.c` program:

```
/* 3.1.2.4.F. windowt◎back_front.c */
/* insert after Delay(180L) */

WindowToBack(FirstWindow);
Delay(180L);
WindowToFront(FirstWindow);
Delay(180L);
```

After a short delay the new window goes behind the large AmigaDOS window then reappears. Here is the format:

```
WindowToBack(Window);
     -306      A0

WindowToFront(Window);
     -312       A0
```

3.1.3 Example window application programs

The first part of this section limited us to viewing structures and window functions which support them. The following sections use example programs to demonstrate how simple it is to use windows under Intuition.

At the beginning we will introduce several general examples which can be used in a variety of applications. Later you'll find window structures and program fragments which were developed for the main project of this book. Many are used in the C program editor that will be developed later on in this book. If you are interested in this excellent editor, enter the basic functions found in this section now. In the following subsections about screens, output, gadgets, etc. you'll find extensions to your basic window program together with descriptions. But you'll need much more than the whole Intuition chapter to complete an editor. The third portion of this section contains functions and routines which deal with various aspects of text editing.

3.1.3.1 All-purpose windows

Each listing here suits a particular range of applications. You can easily adapt the programs to your own needs by making a few changes to the arguments.

About graphic programs

Let's assume that the screen to be drawn on has the appropriate color settings. We'll use the Workbench colors and screen. Section 3.2 shows you how to open your own screens.

Since no operating system functions support drawing graphics on a screen, you have to use windows. The window should resemble a screen as much as possible: maximum size, completely blank and moveable up and down.

Your window can use the full size normally occupied by the screen. For the blank surface, enable the BORDERLESS flag to prevent drawn borders. If the program opens other windows, you will have problems, because these windows may be placed behind your drawing window. You won't be able to access them because your window has no border or gadgets. Omitting the front and back gadgets contradicts good programming sense, nor would it be user-friendly. Simply enable the BACKDROP flag to have a window with the properties of a screen. Here's part of the listing:

```
/* 3.1.3.1.A. GraphicWindow1.c */
struct NewWindow FirstNewWindow =
   {
   0, 0,                        /* LeftEdge, TopEdge    */
   640, 200,                    /* Width, Height        */
   0, 1,                        /* DetailPen, BlockPen */
   NULL,                        /* IDCMP Flags          */
   BACKDROP|                    /* Flags                */
   BORDERLESS |
   SMART_REFRESH,
   NULL,                        /* First Gadget         */
   NULL,                        /* CheckMark            */
   NULL,                        /* Window Title         */
   NULL,                        /* Screen               */
   NULL,                        /* BitMap               */
   0, 0,                        /* Min Width, Height    */
   0, 0,                        /* Max Width, Height    */
   WBENCHSCREEN,                /* Type                 */
   };
```

You won't find a complete listing printed here because it's unnecessary. Simply use this structure in the `First_Window.c` program.

Let's look at the structure values: The position and dimensions correspond to the normal Workbench screen. The window contains no

gadgets and is defined as BACKDROP and BORDERLESS. The refresh mode is set to SMART_REFRESH since SUPER_BITMAP is needed only when graphics are used which are larger than the dimensions of the window. We did not use a title so the title bar won't appear. The result is a completely blank screen which you can use as the drawing surface.

The only problem is the fact that the title bar of the screen may still be visible. It normally overlaps any BACKDROP window, but you don't want it there for a drawing program. You can use `ShowTitle()`, which determines whether or not the title bar of the screen is displayed. As arguments you need a pointer to the screen in question and the set value. The following `main` function performs the task when you add the two functions:

```
/*****************************************/
/*                                       */
/* 3.1.3.1.B. GraphicWindow2.c           */
/*                                       */
/* Program: Window for a graphics program */
/* ====================================  */
/*                                       */
/* Author:  Date:      Comments:         */
/* ------  ----------  ----------        */
/* Wgb     12/06/1987  BACKDROP          */
/* JLD     01/06/1988  BORDERLESS        */
/*                                       */
/*****************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct NewWindow FirstNewWindow =
   {
   0, 0,                    /* LeftEdge, TopEdge    */
   640, 200,                /* Width, Height        */
   0, 1,                    /* DetailPen, BlockPen */
   NULL,                    /* IDCMP Flags          */
/*   WINDOWCLOSE |          /* Flags                */
   BACKDROP |
   BORDERLESS |
   SMART_REFRESH,
   NULL,                    /* First Gadget         */
   NULL,                    /* CheckMark            */
   NULL,                    /*Window Title*/
   NULL,                    /* Screen               */
   NULL,                    /* BitMap               */
   0, 0,                    /* Min Width, Height    */
   0, 0,                    /* Max Width, Height    */
```

```
    WBENCHSCREEN,           /* Type                 */
    };


main()
    {

    Open_All();

    ShowTitle(FirstWindow->WScreen, FALSE);

    Delay(180L);

    Close_All();

    exit(TRUE);
    }


/*************************************
 *                                   *
 * Function: Open library and window *
 * ================================= *
 *                                   *
 * Author:  Date:      Comment:      *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987                *
 *                                   *
 *                                   *
 *************************************/

Open_All()

    {
    void          *OpenLibrary();
    struct Window *OpenWindow();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
       {
       printf("Intuition library not found!\n");
       Close_All();
       exit(FALSE);
       }

    if (!(FirstWindow = (struct Window *)
        OpenWindow(&FirstNewWindow)))
       {
       printf("Window will not open!\n");
       Close_All();
       exit(FALSE);
       }
    }
```

```
/*************************************
 *                                   *
 * Function: Close everything opened *
 * ================================= *
 *                                   *
 * Author:  Date:      Comment:      *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  just Intuition *
 *                     and window    *
 *                                   *
 *************************************/

Close_All()

   {
   if (FirstWindow)      CloseWindow(FirstWindow);
   if (IntuitionBase)    CloseLibrary(IntuitionBase);
   }
```

The next example illustrates the GIMMEZEROZERO window type. The program first opens a simple window. This window closes when you click on the close gadget (you'll finally get to see how to test gadgets).

```
/*************************************
 *                                   *
 * Program: Window GIMMEZEROZERO.C Test*
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16//1987  close with   *
 * JLD     01/09/1989  a Gadget      *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct NewWindow FirstNewWindow =
   {
   160, 50,              /* LeftEdge, TopEdge   */
   320, 150,             /* Width, Height       */
   0, 1,                 /* DetailPen, BlockPen */
   CLOSEWINDOW,          /* IDCMP Flags         */
   WINDOWDEPTH |         /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
```

```
   GIMMEZEROZERO |
   SMART_REFRESH,
   NULL,                  /* First Gadget        */
   NULL,                  /* CheckMark           */
   (UBYTE *)"GIMMEZEROZERO Test",
   NULL,                  /* Screen              */
   NULL,                  /* BitMap              */
   100, 50,               /* Min Width, Height   */
   640, 200,              /* Max Width, Height   */
   WBENCHSCREEN,          /* Type                */
   };


main()
   {
   ULONG  MessageClass;
   USHORT code;

   struct Message *GetMsg();

   Open_All();

   FOREVER
      {
      if (message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort))
         {
         MessageClass = message->Class;
         code = message->Code;
         ReplyMsg(message);
         switch (MessageClass)
            {
            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;
            }
         }
      }
   }


/*************************************
 *                                   *
 * Function: Library and Window open *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16//1989               *
 *                                   *
 *                                   *
 *************************************/

Open_All()

   {
```

```
    void          *OpenLibrary();
    struct Window *OpenWindow();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
       {
       printf("Intuition Library not found!\n");
       Close_All();
       exit(FALSE);
       }

    if (!(FirstWindow = (struct Window *)
        OpenWindow(&FirstNewWindow)))
       {
       printf("Window will not open!\n");
       Close_All();
       exit(FALSE);
       }
    }


/*************************************
 *                                   *
 * Function: Close everthing opened  *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  for Intuition *
 *                     and Window    *
 *                                   *
 *************************************/

Close_All()

    {
    if (FirstWindow)      CloseWindow(FirstWindow);
    if (IntuitionBase)    CloseLibrary(IntuitionBase);
    }
```

Before you compile and run the program, lets talk a bit about reading the system gadgets. In order to get information about the gadget status you need a `Message` structure, defined at the beginning of this program. In addition you need a function to get messages for you. The `GetMsg()` function returns a pointer to a message data structure. You then divide this message into `MessageClass`, the origin of the message, and `Code`. This `Code` is included in the message, but you have no use for it yet. In a switch statement you then test to see if the message is of type `CLOSEWINDOW` (a symbol defined in one of the `include` files) and if so, the window is closed.

This method can be used for much more, but this will be handled in several sections: Section 3.4 (Gadgets), Section 3.7 (Reading the `IDCMP` flags) and Section 3.8.3 (Reading the menus).

Let's expand the first program by having it display the coordinates of the mouse in the DOS window from which you started the program. To read the mouse coordinates you must define two more variables:

```
SHORT Mx,My;
```

Both are assigned values in the FOREVER loop before the `if` test:

```
Mx = FirstWindow->MouseX;
My = FirstWindow->MouseY;
```

And are printed with:

```
printf("X: %d Y: %d\n", Mx, My);
```

After you have run the first version of the program, make the following changes: Add the GIMMEZEROZERO flag to the `NewWindow` structure. Then declare the variables `Gx` and `Gy` as `short` and assign them values after `Mx` and `My` as follows:

```
Gx = FirstWindow->GZZMouseX;
Gy = FirstWindow->GZZMouseY;
```

The `printf` line must also be expanded:

```
printf("X: %d Y: %d ; Gx: %d Gy: %d\n", Mx, My, Gx, Gy);
```

Now when you run the program you will see the difference between the normal and GIMMEZEROZERO windows.

Note the amount of free memory space given in the Workbench display when neither of the programs are running. Then start the first program and write down the memory display value (you will see this when you click somewhere on the Workbench surface with the left mouse button). Then end the first program by clicking on the close gadget. After starting the second program you will notice that it requires more memory than the first.

3.1.3.2 Program routines for text editing

Let's create the windows we could use in the text editor we will create. First we have to consider what we need to assemble a project like a text editor.

You'll need at least three windows for the editor: One for text output, one for file selection and one for requesters and messages.

Editor window The first window allows text output and editing. We'll call this window the editor window.

File selector

The second window will be configured as a file selector box. This type of box lists all filenames in a directory, and lets you select a file using the mouse. Actually this involves a requester, but we will not be discussing gadgets and requesters until later. It is only important that you have the file available so that you can use it.

Message window

We will use the last window as a simple window for messages or requesters. Here again it is important only to have an appropriate file ready.

What properties should the editor window have? It should use a window the size of the entire screen in order to display as much text as possible. Thus, the size of the Workbench screen gives us the maximum window values. You can use any values you like for the minimum, as long as the sizing gadget is still accessible. Finally, we're interested in the various window flags. You should use all the gadgets geared for maximum user-friendliness. We recommend `SMART_REFRESH` mode, since `SUPER_BITMAP` is overkill, and `SIMPLE_REFRESH` will only provide extra work for the program. Here is the `window` structure for our editor:

Structure 3.1: Editor window

```
/* 3.1.3.2.A.editor window */
struct NewWindow EditorWindow =
   {
   0, 0,                        /* LeftEdge, TopEdge    */
   640, 200,                    /* Width, Height        */
   0, 1,                        /* DetailPen, BlockPen */
   NULL,                        /* IDCMP Flags          */
   WINDOWDEPTH |                /* Flags                */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                        /* First Gadget         */
   NULL,                        /* CheckMark            */
   (UBYTE *)"System programming Editor",
   NULL,                        /* Screen               */
   NULL,                        /* BitMap               */
   100, 50,                     /* Min Width, Height    */
   640, 200,                    /* Max Width, Height    */
   WBENCHSCREEN,                /* Type                 */
   };
```

The same parameters set for the editor window apply to the file selector box, except that the file selector box doesn't take up the entire screen. We only include the `WINDOWDEPTH` and `WINDOWDRAG` gadgets because `WINDOWCLOSE` is replaced by another gadget, and sizing would complicate the whole thing even more. Here is this structure:

Structure 3.2: File selector box

```
/* 3.1.3.2.B.filewindow */
struct NewWindow FileWindow =
   {
   180, 25,                  /* LeftEdge, TopEdge   */
   250, 150,                 /* Width, Height       */
   0, 1,                     /* DetailPen, BlockPen */
   NULL,                     /* IDCMP Flags         */
   WINDOWDEPTH |             /* Flags               */
   WINDOWDRAG |
   SMART_REFRESH,
   NULL,                     /* First Gadget        */
   NULL,                     /* CheckMark           */
   (UBYTE *)"File-Select-Box",
   NULL,                     /* Screen              */
   NULL,                     /* BitMap              */
   0, 0,                     /* Min Width, Height   */
   0, 0,                     /* Max Width, Height   */
   WBENCHSCREEN,             /* Type                */
   };
```

The last window which you will always need for the text editor is the message window. It mainly displays error messages, so it is somewhat smaller than the file selector box. In addition, it doesn't contain any system gadgets since it is read and then confirmed, not moved or resized. The window structure is rather simple:

Structure 3.3: Message window

```
/* 3.1.3.2.C.msgwindow */
struct NewWindow MessageWindow =
   {
   250, 100,                 /* LeftEdge, TopEdge   */
   140, 80,                  /* Width, Height       */
   0, 1,                     /* DetailPen, BlockPen */
   NULL,                     /* IDCMP Flags         */
   NULL,                     /* Flags               */
   NULL,                     /* First Gadget        */
   NULL,                     /* CheckMark           */
   NULL,
   NULL,                     /* Screen              */
   NULL,                     /* BitMap              */
   0, 0,                     /* Min Width, Height   */
   0, 0,                     /* Max Width, Height   */
   WBENCHSCREEN,             /* Type                */
   };
```

3.1.3.3 Window for a new CLI

We still need one more window for an editor program.

Our planning begins with writing a small editor which can be used in the CLI instead of the old ED program. Naturally, you need a window for this. What follows is the `main` function with the `window` structure:

Program 3.5: New CLI

```
/*****************************************/
/*                                       */
/* 3.1.3.3.A.    CLIed.c                 */
/*                                       */
/* Program: A new CLI Ed                 */
/* ===================================== */
/*                                       */
/* Author:  Date:      Comments:         */
/* ------  ----------  ----------        */
/* Wgb     10/20//1987 just the window   */
/* JLD     01/09/1989  for testing       */
/*                                       */
/*****************************************/


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

struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct NewWindow ConsoleWindow =
   {
   0, 0,                    /* LeftEdge, TopEdge   */
   640, 101,                /* Width, Height       */
   2, 3,                    /* DetailPen, BlockPen */
   CLOSEWINDOW,             /* IDCMP Flags         */
   WINDOWDEPTH |            /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   ACTIVATE |
   SMART_REFRESH,
   NULL,                    /* First Gadget        */
   NULL,                    /* CheckMark           */
   (UBYTE *)"Wgb Prod. presents  BECKERshell",
   NULL,                    /* Screen              */
   NULL,                    /* BitMap              */
```

```
    640, 50,                /* Min Width, Height   */
    640, 200,               /* Max Width, Height   */
    WBENCHSCREEN,           /* Type                */
    };


main()
    {
    ULONG MessageClass;
    USHORT code;

    struct Message *GetMsg();

    Open_All();

    FOREVER
        {
        if (message = (struct IntuiMessage *)
            GetMsg(FirstWindow->UserPort))
            {
            MessageClass = message->Class;
            code = message->Code;
            ReplyMsg(message);
            switch (MessageClass)
                {
                case CLOSEWINDOW : Close_All();
                                   exit(TRUE);
                                   break;
                }
            }
        }
    }


/*************************************
 *                                   *
 * Function: Open everything         *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/20/1987                *
 *                                   *
 * No parameters                     *
 *                                   *
 *************************************/


Open_All()

    {
    struct Library *OpenLibrary();
    struct Window  *OpenWindow();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
```

```
      {
      printf("Intuition library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&ConsoleWindow)))
      {
      printf("Window can't be opened!\n");
      Close_All();
      exit(FALSE);
      }
   }


/*************************************
 *                                   *
 * Function: Close everything        *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/20/1987                *
 *                                   *
 * No parameters                     *
 *                                   *
 *************************************/


Close_All()

   {
   if (FirstWindow)
      CloseWindow(FirstWindow);

   if (IntuitionBase)
      CloseLibrary(IntuitionBase);
   }
```

3.2 Screen Fundamentals

The last section described many parameters which affect the appearance of windows. Screens play an even more important role in graphic display. Each window parameter depends on the screen to some extent. Window color, resolution and position can be affected by the screen, and base their definitions on the context of the screen.

What purpose does the screen serve in Intuition? The screen contains the essential display parameters. It acts as the background for all visual output. But output has different uses, and there are many different basic assumptions. Let's take two really crude examples:

1.) Word processing's main use for screen output results in text display. As you can see from any book you only need two colors for this (one for text and one for background). It is important that the characters are displayed in high resolution for easy reading.

2.) Multi-color graphics need many colors, as the name suggests. The resolution is important, but some modes don't place as much importance on resolution (such as HAM).

These two examples make it clear that different basic properties have to be available to program. The Amiga goes one step farther here. It allows the programmer to display several different modes using screens, where only one might normally be displayed.

Intuition allows any number of these screens with any number of configuring modes. In principle the procedure is similar to that used for windows. The following pages explain this in more detail.

3.2.1 Creating custom screens

We use a different method for analyzing screen attributes than we used for analyzing windows. You may have become familiar with several types of windows from working with the Workbench. Screens in general are more limited in scope, so the Amiga user may assume that the Workbench screen is all there is. Not so.

As with all operating system routines, structures are used to pass the required information. Therefore, a `NewScreen` structure exists for creating new screens, similar to the `NewWindow` structure for creating windows.

3.2.1.1 The NewScreen structure

Lets take a look at this NewScreen structure and its individual parameters. Here it is:

```
/* 3.2.1.1.A.newscreen_struct */
struct NewScreen
{
0x00  00  SHORT LeftEdge;
0x02  02  SHORT TopEdge;
0x04  04  SHORT Width;
0x06  06  SHORT Height;
0x08  08  SHORT Depth;
0x0a  10  UBYTE DetailPen;
0x00  11  UBYTE BlockPen;
0x0c  12  USHORT ViewModes;
0x0e  14  USHORT Type;
0x10  16  struct TextAttr *Font;
0x14  20  UBYTE *DefaultTitle;
0x18  24  struct Gadget *Gadgets;
0x1c  28  struct BitMap *CustomBitMap;
0x20  32
};
```

The first four arguments have similar meanings as in the NewWindow structure. LeftEdge and TopEdge specify the position of the screen on the monitor. The current versions of Intuition only allow you to change the vertical position, however. LeftEdge is included for compatibility with future versions.

Width and Height let you specify the screen width and height in pixels. You may change the resolution size within limits. For example, you can create a screen only 100 pixels wide but 300 pixels high in non-interlace mode. The maximum values for any screen are 720x400 pixels (720X456 PAL).

The Depth setting of a screen appears as a new argument. Here you set the number of bit-planes allocated for a screen. This determines both the number of colors that can be displayed at once and the amount of memory required. We are interested mainly in the number of colors, calculated according to the formula Colors = 2^Depth.

Let's look at the parameters DetailPen and BlockPen. These determined the colors which Intuition used to draw windows. Carried over to the screen, they set the colors used by Intuition when drawing the title bar, the front gadget and back gadget.

It should be noted that only colors allowed by the Depth can be set. For example, if you set a Depth of three bit-planes, then the maximum number of colors allowed is eight (numbered zero to seven).

Therefore, seven is the highest value for `DetailPen` or `BlockPen`. The same is true for windows, which are completely dependent of the screen settings.

After the pen colors comes a flag by the name of `ViewModes` in the `NewScreen` structure. This is the most important setting—resolution.

As we mentioned before, different types of programs have different resolution requirements. Intuition has a maximum resolution of 320 pixels in the X direction in normal cases. This means that in the rectangular space of your screen there are 320 horizontal points. These pixels are relatively wide and can easily be seen by the eye. In higher resolution the pixel width splits in half, and you have 640 horizontal points.

It's a similar situation in the vertical direction. Normally this is divided into 200 rows in the NTSC version of the Amiga (256 in the PAL version). Here too, you can double the number of lines, but it's accomplished in a different manner. The electron beam of the cathode ray tube writes two sets of information to the screen in alternation, each set shifting slightly vertically from the other.

Here is a summary of all the possible screen settings:

Table 3.3: ViewModes

Flag name	Hex value	Description
HIRES	0x00008000L	Screen with double X resolution
LACE	0x00000004L	Screen with double Y resolution
HAM	0x00000800L	4096 color mode
EXTRA_HALFBRITE	0x00000080L	64 color mode, 5 bit-planes
PFBA	0x00000040L	
DUALPF	0x00000400L	Dual playfield mode
SPRITES	0x00004000L	Sprites can be used
VP_HIDE	0x00002000L	
GENLOCK_AUDIO	0x00000100L	
GENLOCK_VIDEO	0x00000002L	Combines video picture into the graphics through hardware

Before discussing the next value in the structure we would like to clarify the relationship between `Depth` and `ViewModes`. Since the video chip must manage a large amount of memory for your screens, it can become overloaded. You shouldn't be surprised if the video chip can't display more than 16 colors in 640 pixel resolution. With 320 pixels it can display the maximum of 32 colors. If you want to use more colors, you must use the HAM mode instead of the bit-map method.

Screen type

The next structure value sets the screen type. There are only two types: the Workbench screen, opened by the system, and the `CustomScreen` (user screen).

Associated with these two flags are three others which let you enter special features. Intuition must organize memory for the bit-maps. You can do this yourself, however, and set it with another parameter. To inform Intuition that you have already allocated graphic memory, you must set the third flag.

The fourth flag enables you to open the new screen behind all others. This lets you draw a graphic while the screen is hidden without having to first open the screen and then move it to the background.

The fifth flag disables drawing the system gadgets and the title bar. In the program you must then ensure that the right mouse button cannot be used. This can be done by setting a flag. Otherwise the title bar with the menus would be drawn, but since deleting is disabled, it would not be removed.

Here is a summary of the two screen types and the five flags just described:

Table 3.4: Screen type

Flag name	Hex value	Description
WBENCHSCREEN	0x00000001L	Screen type
CUSTOMSCREEN	0x0000000FL	
CUSTOMBITMAP	0x00000040L	Custom bit-map
SCREENBEHIND	0x00000080L	Screen opens in the background
SCREENQUIET	0x00000100L	No gadgets
SHOWTITLE	0x00000010L	(These last two flags set by Intuition itself)
BEEPING	0x00000020L	Screen flashes

Just like each window, a screen has a title bar containing a text string. You can set this title through a window. But if no windows are active, the title bar must remain empty.

Text insertion

The next two parameters allow basic text to be displayed. First, you can use `Font` to point to a `TextAttr` structure, from which you determine the font used for the screen title and all other Intuition text displayed in the screen. If you want to use the font which the user selected with `Preferences`, use the value null (0) instead of a pointer here.

The second value is simply a pointer to a string. If you use null here, no title appears in the title bar.

All we can say about the pointer to the gadget structure is that it was included in the `NewScreen` structure for upward compatibility. Unfor-

tunately, you can't add custom gadgets to a screen. You should always initialize this value as null.

The last pointer normally points to a `BitMap` structure initialized by the program itself. If you don't use this function, you should use the value null here.

3.2.1.2 The first screen listing

Before we go through the `NewScreen` structure in detail as we did for the `NewWindow` structure, you should take a look at the following listing:

Program 3.5: Window on custom screen

```
/*****************************************/
/*                                       */
/* 3.2.1.2.A customscreen_1.c            */
/*                                       */
/* Program: Window on Custom Screen      */
/* ===================================== */
/*                                       */
/* Author: Date:      Comments:          */
/* -----  ----------  ----------         */
/* Wgb    10/16/1987  simple Screen      */
/*                                       */
/*                                       */
/*****************************************/


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


struct IntuitionBase *IntuitionBase;
struct Screen        *FirstScreen;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct NewScreen FirstNewScreen =
   {
   0, 0,                    /* LeftEdge, TopEdge    */
   640, 200,                /* Width, Height        */
   1,                       /* Depth                */
   0, 1,                    /* DetailPen, BlockPen  */
   HIRES,                   /* ViewModes            */
   CUSTOMSCREEN,            /* Type                 */
   NULL,                    /* Font                 */
   (UBYTE *)"Screen Test",
   NULL,                    /* Gadgets              */
   NULL,                    /* CustomBitMap         */
```

```
    };

struct NewWindow FirstNewWindow =
    {
    160, 50,                    /* LeftEdge, TopEdge    */
    320, 150,                   /* Width, Height        */
    0, 1,                       /* DetailPen, BlockPen */
    CLOSEWINDOW,                /* IDCMP Flags          */
    WINDOWDEPTH |               /* Flags                */
    WINDOWSIZING |
    WINDOWDRAG |
    WINDOWCLOSE |
    SMART_REFRESH,
    NULL,                       /* First Gadget         */
    NULL,                       /* CheckMark            */
    (UBYTE *)"Test Custom-Screen",
    NULL,                       /* Screen  */
    NULL,                       /* BitMap               */
    100, 50,                    /* Min Width, Height    */
    640, 200,                   /* Max Width, Height    */
    CUSTOMSCREEN,               /* Type                 */
    };


main()
    {
    ULONG MessageClass;
    USHORT code;

    struct Message *GetMsg();

    Open_All();

    FOREVER
        {
        if (message = (struct IntuiMessage *)
            GetMsg(FirstWindow->UserPort))
            {
            MessageClass = message->Class;
            code = message->Code;
            ReplyMsg(message);
            switch (MessageClass)
                {
                case CLOSEWINDOW : Close_All();
                                   exit(TRUE);
                                   break;
                }
            }
        }
    }
```

```
/*******************************************
 *                                         *
 * Function: Open library, screen and window *
 * ======================================= *
 *                                         *
 * Author:  Date:      Comments:           *
 * ------  ----------  ----------          *
 * Wgb     10/16/1987                      *
 *                                         *
 *                                         *
 *******************************************/

Open_All()
   {
   void          *OpenLibrary();
   struct Window *OpenWindow();
   struct Screen *OpenScreen();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition library cannot be found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstScreen = (struct Screen *)
       OpenScreen(&FirstNewScreen)))
      {
      printf("Screen cannot be opened!\n");
      Close_All();
      exit(FALSE);
      }

   FirstNewWindow.Screen = FirstScreen;

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&FirstNewWindow)))
      {
      printf("Window cannot be opened!\n");
      Close_All();
      exit(FALSE);
      }
   }

/*************************************
 *                                   *
 * Function: Close everything opened *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  Window, screen *
 *                     and intuition *
 *                                   *
 *************************************/
```

```
Close_All()
   {
   if (FirstWindow)      CloseWindow(FirstWindow);
   if (FirstScreen)      CloseScreen(FirstScreen);
   if (IntuitionBase)    CloseLibrary(IntuitionBase);
   }
```

How it works You should be familiar with the structure of this listing. After the general pointer and structure definitions comes the `main` function which calls the `OpenAll()` function, then branches to a loop which waits until you click on the close gadget. The `CloseAll()` function ends the program. Let's look at the four main sections:

Among the pointers we find one for screen structure. It places the window later. First you need the `NewScreen` structure, which opens a simple screen which has the same properties as the Workbench screen, with the exception that you request only two colors (i.e., one bitplane).

You should recognize the `main` program since we explained how to read the close gadget at the end of the last section.

The `Open_All()` function includes a screen opening routine. The order of opening is as follows: Library (no library, no `OpenScreen()` function), screen, window. A pointer to the custom screen appears in the window.

The `Close_All()` function closes the window first, then the screen, then the library. Everything occurs in the reverse of the order of the opening process.

If you like, you can change one or two of the arguments of the `NewWindow` or `NewScreen` structure. Don't make too many changes at once though, since it's easy to overlook an error.

3.2.1.3 The `Screen` structure

After this programming interlude, let's return to the `Screen` structure set up by Intuition. It contains many values, pointers and flags. Here is the structure, complete with offsets:

```
/* 3.2.1.3.A. screenstructure */
struct Screen
{
0x00  00  struct Screen *NextScreen;
0x04  04  struct Window *FirstWindow;
0x08  08  SHORT LeftEdge;
0x0A  10  SHORT TopEdge;
```

```
0x0C  12  SHORT Width;
0x0E  14  SHORT Height;
0x10  16  SHORT MouseY;
0x12  18  SHORT MouseX;
0x14  20  USHORT Flags;
0x16  22  UBYTE *Title;
0x1A  26  UBYTE *DefaultTitle;
0x1E  30  BYTE BarHeight;
0x1F  31  BYTE BarVBorder;
0x20  32  BYTE BarHBorder;
0x21  33  BYTE MenuVBorder;
0x22  34  BYTE MenuHBorder;
0x23  35  BYTE WBorTop;
0x24  36  BYTE WBorLeft;
0x25  37  BYTE WBorRight;
0x26  38  BYTE WBorBottom;
0x27  39  struct TextAttr *Font;
0x2B  43  struct ViewPort ViewPort;
    {
    0x00  00  struct ViewPort *Next;
    0x04  04  struct ColorMap *ColorMap;
    0x08  08  struct CopList *DspIns;
    0x0C  12  struct CopList *SprIns;
    0x10  16  struct CopList *ClrIns;
    0x14  20  struct UCopList *UCopIns;
    0x18  24  SHORT DWidth;
    0x1A  26  SHORT DHeight;
    0x1C  28  SHORT DxOffset;
    0x1E  30  SHORT DyOffset;
    0x20  32  UWORD Modes;
    0x22  34  UBYTE SpritePriorities;
    0x23  35  UBYTE reserved;
    0x24  36  struct RasInfo *RasInfo;
    0x28  40
    };
0x53  83  struct RastPort RastPort;
    {
    0x00  00  struct Layer *Layer;
    0x04  04  struct BitMap *BitMap;
    0x08  08  USHORT *AreaPtrn;
    0x0C  12  struct TmpRas *TmpRas;
    0x10  16  struct AreaInfo *AreaInfo;
    0x14  20  struct GelsInfo *GelsInfo;
    0x18  24  UBYTE Mask;
    0x19  25  BYTE FgPen;
    0x1A  26  BYTE BgPen;
    0x1B  27  BYTE AOlPen;
    0x1C  28  BYTE DrawMode;
    0x1D  29  BYTE AreaPtSz;
    0x1E  30  BYTE linpatcnt;
    0x1F  31  BYTE dummy;
    0x20  32  USHORT Flags;
    0x22  34  USHORT LinePtrn;
    0x24  36  SHORT cp_x;
    0x26  38  SHORT cp_y;
    0x28  40  UBYTE minterms[8];
```

```
    0x30  48  SHORT PenWidth;
    0x32  50  SHORT PenHeight;
    0x34  52  struct TextFont *Font;
    0x38  56  UBYTE AlgoStyle;
    0x39  57  UBYTE TxFlags;
    0x3A  58  UWORD TxHeight;
    0x3C  60  UWORD TxWidth;
    0x3E  62  UWORD TxBaseline;
    0x40  64  WORD TxSpacing;
    0x42  66  APTR *RP_User;
    0x46  70  ULONG longreserved[2];
    0x4E  78  UWORD wordreserved[7];
    0x5C  92  UBYTE reserved[8];
    0x64  100
    };
0xB7 183  struct BitMap BitMap;
    {
    0x00  00  UWORD BytesPerRow;
    0x02  02  UWORD Rows;
    0x04  04  UBYTE Flags;
    0x05  05  UBYTE Depth;
    0x06  06  UWORD pad;
    0x08  08  PLANEPTR Planes[8];
    0x28  40
    };
0xDF 223  struct Layer_Info LayerInfo;
    {
    0x00  00  struct Layer *top_layer;
    0x04  04  struct Layer *check_lp;
    0x08  08  struct Layer *obs;
    0x0C  12  struct MinList FreeClipRects;
    0x18  24  struct SignalSemaphore Lock;
    0x3C  60  struct List gs_Head;
    0x4A  74  LONG longreserved;
    0x4E  78  UWORD Flags;
    0x50  80  BYTE fatten_count;
    0x51  81  BYTE LockLayersCount;
    0x52  82  UWORD LayerInfo_extra_size;
    0x54  84  WORD *blitbuff;
    0x58  88  struct LayerInfo_extra *LayerInfo_extra;
    0x5C  92
    };
0x13B 315 struct Gadget *FirstGadget;
0x13F 319 UBYTE DetailPen
0x140 320 UBYTE BlockPen;
0x141 321 USHORT SaveColor0;
0x143 323 struct Layer *BarLayer;
0x147 327 UBYTE *ExtData;
0x14B 331 UBYTE *UserData;
0x14F 335
};
```

*NextScreen Like the windows, all open screens are linked together. The link pointer resides in this variable.

*FirstWindow
: Each screen contains the pointer of the first window opened. The windows are then linked together.

LeftEdge, TopEdge, Width and Height
: These four values were explained in conjunction with the NewScreen structure above.

MouseX, MouseY
: As with the windows, these variables contain the mouse coordinates relative to your screen.

Flags
: This value is initialized as in the NewScreen structure, with two additional flags:

SHOWTITLE
: This flag is set when the title bar is displayed with ShowTitle().

BEEPING
: This flag is set when the screen flashes with DisplayBeep().

*Title
: As with the windows, this is a pointer to the title text string.

*DefaultTitle
: This pointer comes into play when a window is opened without a specific title. Intuition then uses the DefaultTitle.

The following nine variables of the Screen structure apply to all windows in the screen as well as the screen itself:

BarHeight
: Title bar height, measured in screen lines.

BarVBorder
: Vertical border width.

BarHBorder
: Horizontal border width.

MenuVBorder
: Vertical menu border width.

MenuHBorder
: Horizontal menu border width.

WBorTop
: Top window border width.

WBorLeft
: Left window border width.

WBorRight
: Right window border width.

WBorBottom
: Bottom window border width.

*Font
: Pointer to the default font to be used by Intuition. Initialized by the NewScreen structure.

`ViewPort` This structure contains information about the display needed for the Copper, the video chip.

`RastPort` This structure manages the screen and contains all information required for drawing on the screen.

`BitMap` This structure contains more precise information about the bit-map. It also contains pointers to each bit-map.

`LayerInfo` Layers are the basis of the window management. This information structure is the beginning.

`*FirstGadget`
Pseudo-pointer to the screen custom gadgets (not implemented).

`DetailPen,BlockPen`
The two drawing pens, as explained for `NewScreen` and `NewWindow`.

`SaveColor()` Since the background color changes when a screen flashes, this location acts as a buffer for color 0.

`*BarLayer` This pointer points to the layer (graphic storage area) in which the title bar of the screen is stored.

`*ExtData` Pointer to external data which can be added.

`*UserData` Pointer to data managed by the user.

3.2.1.4 The screen functions

As you learned in the example programs, there are also open and close functions for screens, called `OpenScreen()` and a `CloseScreen()`. The arguments are also similar: the pointer to the `NewScreen` structure for the former, and the structure pointer returned by Intuition for the latter.

As you may have guessed, the user has quite a number of screen functions available for him/her. Let's take a look at them.

For instance, Intuition functions support clicking on the front and back gadgets. With `ScreenToFront()` and `ScreenToBack()` you can move any screen to which you have the pointer to the foreground or background. To demonstrate this, remove the gadget test from the screen example program listed above. The `main` function should then look like this:

```
main()
   {

   Open_All();

   Delay(180L);

   Close_All();
   exit(TRUE);
   }
```

The `Delay()` line must also be inserted so that you notice that something is happening when the new screen opens. Add all of the extensions after the `Delay(180L);` line unless told otherwise. First an example of `ScreenToBack()` and `ScreenToFront()`:

```
ScreenToBack(FirstScreen);
Delay(180L);
ScreenToFront(FirstScreen);
```

This very simple example moves the screen to the background and the foreground after a short delay, but it shows you how to use the functions. Here is the general syntax with the arguments, registers and function offset:

```
ScreenToBack(Screen);
     -246       a0

ScreenToFront(Screen);
     -252       A0
```

The same possibilities which can be used for a custom screen can also be used for the Workbench screen, of course. Normally the problem is that you don't have a pointer to the Workbench screen. Another function does this without using pointers:

```
WBenchToBack();
    -336

WBenchToFront();
    -342
```

Insert these three lines:

```
WBenchToFront();
Delay(180L);
WBenchToBack();
```

You may be familiar with the following situation in BASIC: You try to move out of the text area with the cursor in the editor and the screen flashes and beeps. Intuition sends this flash as a warning. The following routine produces five warning signals in a row:

```
for (i=1; i<6; i++)
   {
   for (j=0; j<10000; j++);
   DisplayBeep(FirstScreen);
   }
```

To start the new program, first declare the two loop variables `i` and `j` as `int`. The program then flashes the new custom screen five times.

If you want to flash all of the screens (e.g., if you don't know what screen is currently in the foreground), then null can be used as the pointer to tell Intuition that it should flash all of the screens:

```
DisplayBeep(NULL);
```

Here is the format of the function call:

```
DisplayBeep(Screen);
     -96       A0
```

No WorkBench

Many applications may run out of graphic memory very quickly. This cannot be corrected by expanding the memory because the chip RAM area cannot be expanded beyond 512K.

The Amiga developers decided that a program would not need the Workbench screen if it opened a screen of its own. In these cases you can close the Workbench screen. The only condition: No other programs can be running which display data on a window in the Workbench screen. If this condition is met, Intuition lets you close the Workbench window with `CloseWorkBench()`.

How can you start a program without using the Workbench? Unfortunately you cannot start your program from the CLI window because the CLI is an independent program which outputs to the Workbench screen. Therefore we will give the program an icon so that it can be called by double-clicking on the icon.

Add the following lines to `CustomScreen1.c` program and save it to disk as `CloseBench.c`. Compile the new program, and remember to link it with `startup.o`. (You may have to compile `startup.h`). At the end of the `Open_All()` function after opening the window add:

```
CloseWorkBench();
```

At the start of the `Close_All()` function add:

```
OpenWorkBench();
```

Next copy the `CLI.INFO` icon to `CloseBench.INFO`. Close all programs that you started from the workbench, the CLI is a running program, and start the `CloseBench` program. When you pull the screen down you will notice that there is no longer a Workbench screen behind it.

Moving Screens

Each screen whose title bar is not hidden by a BACKDROP window can be moved vertically by the user. As with windows, an Intuition function allows screens to be moved. This is no problem with MoveScreen() and the corresponding delta values.

We have prepared two lines which move the screen back up if you move it down. Insert the following lines in CustomScreen1.c program in the FOREVER loop before the message test:

```
if (FirstScreen->TopEdge != 0)
   MoveScreen(FirstScreen, 0L, -1L);
```

The general format of MoveScreen() is:

```
MoveScreen(Screen, DeltaX, DeltaY);
    -162      A0       D0       D1
```

Since the ShowTitle() function was explained in the window section, just two functions remain which deal with screens. The first is GetScreenData(), which returns a summary of screen data.

All of the values could be read through a simple structure access, but then you don't get a summary at a particular time because all of the parameters could change as you read them. Therefore you set up a buffer for this function in which the data are stored. Then you call the function and all of the arguments transfer to your buffer. This is especially useful if you want information about the Workbench screen. Since you can specify the type of the window with a flag, it is easy to set the Workbench type. This allows you to easily acquire normally inaccessible data. Since an example is rather difficult to formulate, here is the syntax:

```
Success = GetScreenData(Buffer, Size, Type, Screen);
  D0           -426        A0      D0     D1      A1
```

MakeScreen() lets you manipulate a screen managed by Intuition. After calling MakeScreen() and passing it a screen pointer, it waits until Intuition View is no longer needed, performs its task using MakeVPort() and releases Intuition View.

Here's the format of MakeScreen():

```
MakeScreen(Screen);
   -378       A0
```

These are all the Intuition functions which control screens. We hope the examples helped you understand the functions.

3.2.2 Examples of using screens

In conclusion we would like to present some short programs and extended the program sections which you can use in your existing programs. You'll find that some of these program excerpts are intended for the "big editor project" later on in this book.

Unfortunately, we don't have all that many examples. This is because screens are required only for special applications.

3.2.2.1 General examples

In this section we present two screen examples in the subject of graphic programs. These examples may give you some ideas for your own programming.

First is a super-high-resolution screen with just one bit-plane. Super-high-resolution means a resolution of 640x400 pixels on an NTSC monitor (or 640x512 on a PAL monitor). We chose just one bit-plane because we use enough memory in the higher resolution. You must remember that you only have 512K of graphic memory available (CHIP_MEMORY).

Structure 3.3: Super screen

```
/* 3.2.2.1.A. superscreen struct */

struct NewScreen SuperScreen =
   {
   0, 0,                  /* LeftEdge, TopEdge   */
   640, 400,              /* Width, Height       */
   1,                     /* Depth               */
   0, 1,                  /* DetailPen, BlockPen */
   HIRES |                /* ViewModes           */
   LACE,
   CUSTOMSCREEN |         /* Type                */
   SCREENQUIET,
   NULL,                  /* Font                */
   NULL,
   NULL,                  /* Gadgets             */
   NULL,                  /* CustomBitMap        */
   };
```

This mode is suited for digitized pictures and for super hi-res graphics. Here is the complete example program:

```
/*************************************
 *                                   *
 * Program: Superscreen.c            *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/87    one Screen    *
 *                                   *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Screen        *FirstScreen;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct NewScreen SuperScreen =
   {
   0, 0,                /* LeftEdge, TopEdge   */
   640, 400,            /* Width, Height       */
   1,                   /* Depth               */
   0, 1,                /* DetailPen, BlockPen */
   HIRES |              /* ViewModes           */
   LACE,
   CUSTOMSCREEN |       /* Type                */
   SCREENQUIET,
   NULL,                /* Font                */
   NULL,
   NULL,                /* Gadgets             */
   NULL,                /* CustomBitMap        */
   };


struct NewWindow FirstNewWindow =
   {
   160, 50,             /* LeftEdge, TopEdge   */
   320, 150,            /* Width, Height       */
   0, 1,                /* DetailPen, BlockPen */
   CLOSEWINDOW,         /* IDCMP Flags         */
   WINDOWDEPTH |        /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                /* First Gadget        */
   NULL,                /* CheckMark           */
   (UBYTE *)"Test Custom-Screen",
   NULL,                /* Screen              */
   NULL,                /* BitMap              */
```

```
    100, 50,            /* Min Width, Height   */
    640, 150,           /* Max Width, Height   */
    CUSTOMSCREEN,       /* Type                */
    };


main()
    {
    Open_All();

    Delay(180L);

    Close_All();
    exit(TRUE);
    }


/***********************************************
 *                                             *
 * Function: Library, Screen and Window open   *
 * ========================================== *
 *                                             *
 ***********************************************/

Open_All()
    {
    void          *OpenLibrary();
    struct Window *OpenWindow();
    struct Screen *OpenScreen();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
       {
       printf("Intuition Library not found!\n");
       Close_All();
       exit(FALSE);
       }

    if (!(FirstScreen = (struct Screen *)
        OpenScreen(&SuperScreen)))
       {
       printf("Screen not opened, no page!\n");
       Close_All();
       exit(FALSE);
       }

    FirstNewWindow.Screen = FirstScreen;

    if (!(FirstWindow = (struct Window *)
        OpenWindow(&FirstNewWindow)))
       {
       printf("Window will not open!\n");
       Close_All();
       exit(FALSE);
       }
    }
```

```
/*************************************
 *                                   *
 * Function: Close everything opened *
 * ================================= *
 *                                   *
 *************************************/

Close_All()
   {
   if (FirstWindow)      CloseWindow(FirstWindow);
   if (FirstScreen)      CloseScreen(FirstScreen);
   if (IntuitionBase)    CloseLibrary(IntuitionBase);
   }
```

Partial screens In the second example we want to show you that you don't have to use the full resolution. An interlace screen can also be initialized with 200 lines. It then takes up only half the monitor. It should be mentioned that any other screen used along with an interlace screen also flickers.

Structure 3.4: Utility screen

```
/* 3.2.2.1.B. screen util */
struct NewScreen Utility =
   {
   0, 150,                    /* LeftEdge, TopEdge    */
   640, 50,                   /* Width, Height        */
   2,                         /* Depth                */
   2, 1,                      /* DetailPen, BlockPen */
   HIRES,                     /* ViewModes            */
   CUSTOMSCREEN,              /* Type                 */
   NULL,                      /* Font                 */
   (UBYTE *)"Utility-Screen",
   NULL,                      /* Gadgets              */
   NULL,                      /* CustomBitMap         */
   };
```

The structure presented here opens a screen in the lower 50 lines of the Workbench screen. You might want to open a BORDERLESS window here in which you could place gadgets for calling up various utility functions. Be sure to adjust your window size so it fits on the screen.

3.2.2.2 Program sections for the text editor

You may want to use your C source code editor while the compiler is running, or you may want it removed completely from the screen if, for example, there are too many windows open on the Workbench and the processing time takes too long. In that case, it's a good idea for a new program to open a new screen. Then you can exit the program and work on the Workbench undisturbed.

The structure presented here is configured especially for text editing. It uses only one bit-plane to use as little memory as possible. This is

enough, since text doesn't rely heavily on colors. High-resolution makes the text easily readable.

Structure 3.5: Editor screen

```
/* 3.2.2.2.A. edscreen */
struct NewScreen EditorScreen =
   {
   0, 0,                          /* LeftEdge, TopEdge   */
   640, 200,                      /* Width, Height       */
   1,                             /* Depth               */
   1, 0,                          /* DetailPen, BlockPen */
   HIRES,                         /* ViewModes           */
   CUSTOMSCREEN,                  /* Type                */
   NULL,                          /* Font                */
   (UBYTE *)"Abacus C Editor Screen",
   NULL,                          /* Gadgets             */
   NULL,                          /* CustomBitMap        */
   };
```

3.3 Output

After having completed the first two sections of this chapter, you now know almost every capability of Intuition screens and windows; how they open and close; and how other functions interact with them.

But what use is a screen or window that appears on the screen and does nothing? Except, perhaps, letting the user move it up or down.

At the beginning of this book we mentioned that windows are the basis for all input and output. The screens simply exist to subdivide areas and specify default properties of the windows. You know how to set up screens and windows, but how do you control input and output? This section discusses the subject of output.

The Amiga offers two types of output. The first type comes from the output functions found in `graphics.library`. These are the basis for all of the screen output functions. The second type is supported by `intuition.library`. It offers only three output functions, but they are flexible enough that all other elements can be constructed from them. All user elements such as gadgets, requesters, and menus use the three graphic functions supplied by Intuition.

You see how important it is to explore these functions. Later you will see how easy it is to construct the most complex objects using the building block system of Intuition.

In addition, you'll get to use the windows for a real application.

3.3.1 Text output

We'll go through the individual output options step by step. Let's look at text output first, since it's the most important form of communication with the user.

When starting out, it is important to know the parameters your text can have. They are easy to understand.

3.3.1.1 Position, color, character mode

About the position: You must determine the exact pixel at which your text is written relative to your window. In addition, you can set the color used for the background and foreground (the text lines themselves). Here you access a certain number of color pens which the screen offers you. Pen number -1 (0xFF) plays a special role. This number represents the window's default color.

As you are probably aware, the operating system has various modes available for producing output.

JAM1 The first mode takes into account what is drawn but not the background. This displays the text lines only, without disturbing the background. This mode is called `JAM1` because color 1 is "jammed" into the graphic.

JAM2 In the second mode Intuition displays both the characters and the background. This mode is called `JAM2` because both colors are drawn.

INVERSEVID The third mode is similar to the second except that colors are exchanged. You need `INVERSEVID` for the default colors, however, which are designated with -1 (0xFF) and cannot be simply exchanged.

COMPLEMENT Mode number four, `COMPLEMENT`, is similar to one except that the background is drawn correspondingly. Set points are erased and erased points are set.

Here are the four modes:

Table 3.5: DrawModes

Drawing mode	Description
JAM1	Only pen 1 used
JAM2	Both pens used
INVERSEVID	Like JAM2, with pens exchanged
COMPLEMENT	Like JAM1, set points cleared, cleared set

With the last text parameter you can specify a character font as a `TextAttr` structure.

3.3.1.2 The `IntuiText` structure

As usual, the Intuition parameters are stored in a structure, which looks like this:

```
/* 3.3.1.2.A. intuitext_struct */
struct IntuiText
{
0x00  00  UBYTE FrontPen;
0x01  01  UBYTE BackPen;
0x02  02  UBYTE DrawMode;
0x03  03  SHORT LeftEdge;
0x05  05  SHORT TopEdge;
0x07  07  struct TextAttr *ITextFont;
0x0B  11  UBYTE *IText;
0x0F  15  struct IntuiText *NextText;
0x13  19
};
```

The last two elements of the structure need some explaination. `IText` represents a pointer to a string terminated by null. With the pointer `NextText`, the system makes it possible to link several such `IntuiText` structures together into a chain. The advantage of this lies in the call to produce the output. This capability means that you aren't limited to a single line or color setting per call. This allows maximum flexibility with just one function call.

3.3.1.3 Fonts and type styles

Any of the fonts on the Workbench disk can be used, but Intuition offers another option.

A default font can be set for any window. If you want to use this font (generally set in Preferences), then you just use null instead of a pointer.

You can also access a custom font through the `diskfont.library`. You should usually use the two ROM fonts, however, to retain readability and consistency. These fonts are always accessible and have uniform dimensions.

To select the two ROM fonts you first need the `TextAttr` structure, which defines both the font and its style. This structure looks like this:

```
/* 3.3.1.3.A. textattributes */
struct TextAttr
{
0x00  00  STRPTR ta_Name;
0x04  04  UWORD ta_YSize;
0x06  06  UBYTE ta_Style;
0x07  07  UBYTE ta_Flags;
0x08  08
};
```

ta_Name contains a pointer to a string which states the font name. If you want to use the ROM fonts, insert topaz.font. Make sure that the font name is entirely in lowercase letters or the operating system will not recognize it. If you have opened other fonts with the diskfont.library, you can access them as well.

With the next value you set the height of the font in pixels. This determines the size of the font and also the number of characters per line. ROM fonts have two sizes:

Table 3.6: Standard fonts

Font size	Characters	Height
TOPAZ_SIXTY	Sixty	At 640 pixels = 9 pixels
TOPAZ_EIGHTY	Eighty	At 640 pixels = 8 pixels.

The next element in the TextAttr structure, ta_Style, allows you to set something called the soft-style. This lets you change the appearance of a normal font by applying certain functions to it. Using this you can implement underlining, italics, boldface and extended characters. The following flags are available and can be freely combined:

Table 3.7: Type styles

Flag name	Hex value	Meaning
FS_NORMAL	0x00	No effect
FSF_ITALIC	0x04	Italic
FSF_BOLD	0x02	Boldface
FSF_UNDERLINED	0x01	Underline
FSF_EXTENDED	0x08	Extended characters

In the last value of the structure you use flags to specify additional information about your character set. These flags are called font preference flags. We are interested in just two of these flags—the two which indicate from where Intuition should get the font. You can use ROM fonts or disk fonts. Since topaz.font is in ROM, you set the corresponding flag:

Table 3.8: Font type

Flag name	Hex value	Meaning
FPF_ROMFONT	0x01	Get font from ROM
FPF_DISKFONT	0x02	Get font from disk
FPF_REVPATH	0x04	
FPF_TALLDOT	0x08	
FPF_WIDEDOT	0x10	
FPF_PROPORTIONAL	0x20	
FPF_DESIGNED	0x40	
FPF_REMOVED	0x80	

To use topaz.font with a line length of 80 characters and normal appearance, the structure would look like this:

```
/*3.3.1.3.B. testfont */
struct TextAttr TestFont =
   {
   (STRPTR)"topaz.font",
```

```
TOPAZ_EIGHTY,
FS_NORMAL,
FPF_ROMFONT
};
```

3.3.1.4 PrintIText

Now that you've seen the individual parameters, it's time to work with the IntuiText structure.

The PrintIText function is required for displaying the IntuiText. In addition, you need an argument from the window in which the output appears. Your argument is the RastPort, without this it is impossible to perform any output.

You can do this with the following definition and assignment:

```
struct RastPort *MyWindowRastPort;
MyWindowRastPort = MyWindow->RPort;
```

Now you need an appropriate IntuiText structure to start output. We've written one for you:

```
/* 3.3.1.4.A.firsttext */
struct IntuiText FirstText =

   {
   1, 0,                        /* FrontPen, BackPen    */
   JAM2,                        /* DrawMode             */
   15, 0,                       /* LeftEdge, TopEdge    */
   NULL,                        /* Font (Standard)      */
   (UBYTE *) "System programming on the Amiga!",
   NULL                         /* NextText             */
   };
```

Insert this definition in the Customscreen1.c program. Remember that this definition must precede the main() function.

RastPort can be determined in the main() function after Open_All() finishes with the assignment statement given above.

You are almost ready to call PrintIText(). All you need yet are coordinates.

You may wonder why you need them. Take a look at the following graphic:

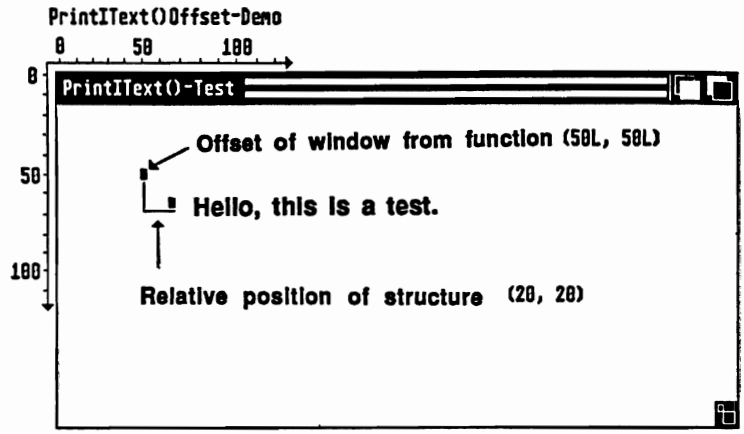


Figure 3.3

When you call the `PrintIText()` function, you determine the point relative to the `RastPort` to which all other parameters in the structures refer. This point becomes the origin. The position specified in the `IntuiText` structure then relates to this. Insert the following line in your `Customscreen1.c` program:

```
PrintIText(MyWindowRastPort, FirstText, 10L, 20L);
```

Note: Be sure to specify the offsets of `PrintIText()` as `long` values or the Aztec C compiler may have problems. Here is the complete listing:

```
/*************************************
 *                                   *
 * Program: PrintIText.c             *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  PrintIText Test *
 *                     Window        *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;


struct NewWindow FirstNewWindow =
```

```
    {
    160, 50,              /* LeftEdge, TopEdge    */
    320, 150,             /* Width, Height        */
    0, 1,                 /* DetailPen, BlockPen */
    NULL,                 /* IDCMP Flags          */
    WINDOWDEPTH |         /* Flags                */
    WINDOWSIZING |
    WINDOWDRAG |
    WINDOWCLOSE |
    SMART_REFRESH,
    NULL,                 /* First Gadget         */
    NULL,                 /* CheckMark            */
    (UBYTE *)"System programimg test",
    NULL,                 /* Screen               */
    NULL,                 /* BitMap               */
    100, 50,              /* Min Width, Height    */
    640, 200,             /* Max Width, Height    */
    WBENCHSCREEN,         /* Typ                  */
    };

struct IntuiText FirstText =
    {
    1, 0,                 /* FrontPen, BackPen    */
    JAM2,                 /* DrawMode             */
    15, 0,                /* LeftEdge, TopEdge    */
    NULL,                 /* Font (Standard)      */
    (UBYTE *) "System programing on the Amiga!",
    NULL                  /* NextText             */
    };



main()
    {
    struct RastPort *MyWindowsRastPort;

    Open_All();

    MyWindowsRastPort = FirstWindow->RPort;

    PrintIText(MyWindowsRastPort, &FirstText, 10L, 20L);

    Delay(180L);

    Close_All();
    }


/*************************************
 *                                   *
 * Function: Library and Window open *
 * ================================= *
 *                                   *
 *************************************/

Open_All()
```

```
    {
    void          *OpenLibrary();
    struct Window *OpenWindow();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
        {
        printf("Intuition Library not found!\n");
        Close_All();
        exit(FALSE);
        }

    if (!(FirstWindow = (struct Window *)
        OpenWindow(&FirstNewWindow)))
        {
        printf("Window will not open!\n");
        Close_All();
        exit(FALSE);
        }
    }


/*************************************
 *                                   *
 * Function: Close All               *
 * ================================= *
 *                                   *
 *************************************/

Close_All()

    {
    if (FirstWindow)      CloseWindow(FirstWindow);
    if (IntuitionBase)    CloseLibrary(IntuitionBase);
    }
```

PrintIText format:

```
PrintIText(RastPort, IText, LeftOffset, TopOffset);
    -216       A0        A1       D0          D1
```

Linked text

To round everything out, here is an example of a linked `IntuiText` structure, which you can print by calling `PrintIText` as above. The system takes care of whether or not multiple strings exist, and displays everything correctly. Note that when defining these structures in this manner the last text in the list must be defined first.

```
struct IntuiText FourthText =
    {
    1, 0,                 /* FrontPen, BackPen    */
    JAM2,                 /* DrawMode             */
    15, 20,               /* LeftEdge, TopEdge    */
```

```
    NULL,                   /* Font (Standard)      */
    (UBYTE *) "This text is an example ",
    NULL                    /* NextText             */
    };

struct IntuiText ThirdText =
    {
    0, 1,                   /* FrontPen, BackPen    */
    JAM2,                   /* DrawMode             */
    15, 30,                 /* LeftEdge, TopEdge    */
    NULL,                   /* Font (Standard)      */
    (UBYTE *) "of linking ",
    &FourthText            /* NextText             */
    };

struct IntuiText SecondText =
    {
    2, 0,                   /* FrontPen, BackPen    */
    JAM2,                   /* DrawMode             */
    15, 40,                 /* LeftEdge, TopEdge    */
    NULL,                   /* Font (Standard)      */
    (UBYTE *) "multiple strings ",
    &ThirdText             /* NextText             */
    };

struct IntuiText FirstText =
    {
    3, 0,                   /* FrontPen, BackPen    */
    JAM2,                   /* DrawMode             */
    15, 50,                 /* LeftEdge, TopEdge    */
    NULL,                   /* Font (Standard)      */
    (UBYTE *) "with IntuiText.",
    &SecondText            /* NextText             */
    };
```

IntuiText structure

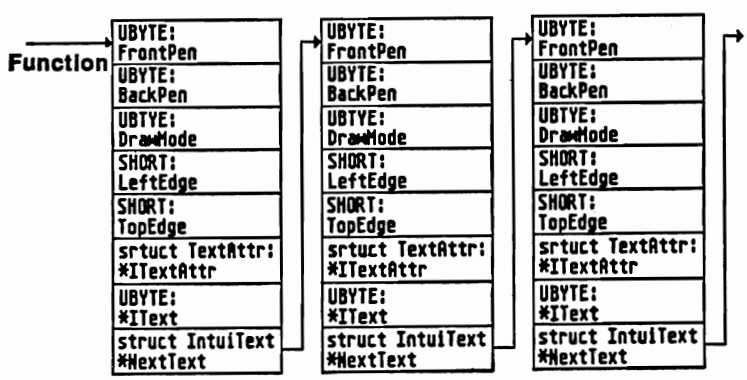


Figure 3.4

3.3.2 Drawing lines

The second output element which Intuition supports is line output. You may be familiar with the importance of these lines if you have ever seen a Workbench requester. Each gadget has a double border surrounding it. The scroll bars in the directory windows are also framed with a box made up of these lines.

First we'll examine the parameters in the `Border` structure so that you can draw simple lines.

3.3.2.1 Color, position, etc.

As with the `IntuiText` structure, the `Border` structure can be used to establish the starting position of the first line to be drawn. As you saw with `PrintIText()`, an offset is used as the relative origin for the position specified in the function call.

`FrontPen` and `BackPen` can also be set, although `BackPen` is not used (a line really doesn't have a defined background). For this same reason you can't use all four modes provided for `DrawMode`. Only `JAM1` and `INVERSVID` make sense because the other two refer to the second color (again, an unused color in this case).

3.3.2.2 The `Border` structure

Let's look at the `Border` structure before going on to the parameters:

```
/* 3.3.2.2.A. borderstruct */
struct Border
{
0x00  00  SHORT LeftEdge
0x02  02  SHORT TopEdge;
0x04  04  UBYTE FrontPen
0x05  05  UBYTE BackPen;
0x06  06  UBYTE DrawMode;
0x07  07  BYTE Count;
0x08  08  SHORT *XY;
0x0C  12  struct Border *NextBorder;
0x10  16
};
```

Unlike the `IntuiText` structure, no definition of a font exists for the border. But you need a value which tells the types of corner points your sequence of lines will have.

Correspondingly you also have a pointer to tables of coordinate pairs instead of a string. `Count` specifies the number of pairs in this table.

As with texts, multiple border structures can be linked together to perform tasks like change line colors.

3.3.2.3 Coordinate table for line drawing

The number of coordinate pairs in the table is stored in the structure in the variable `Count`. You have to supply a pointer to the first element.

A coordinate table consists of `short` value pairs; one value for the X position and one for the Y position. These values are viewed as offsets from `LeftEdge` and `TopEdge`, which you initialized in the `Border` structure.

To complicate matters, we should mention that these coordinates are just offsets to the coordinates of the `DrawBorder()` function. The following graphic makes this clear:

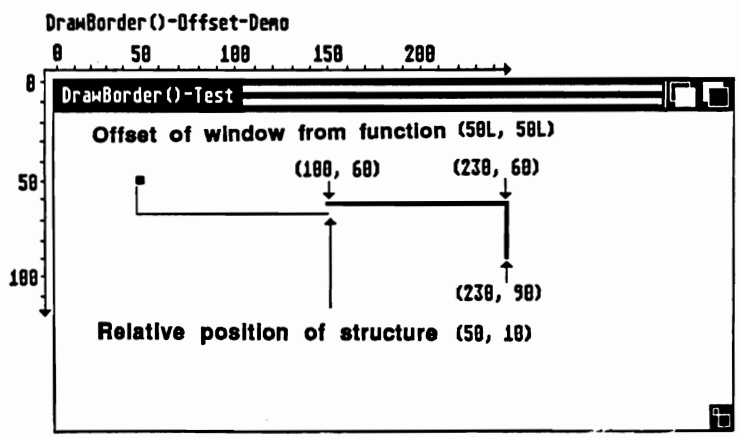


Figure 3.5

A coordinate table could look like this:

```
SHORT TestValues[] =
   {
    0,  0,
   50,  0,
   50, 12,
```

```
    0, 12,
    0,  0
   };
```

This table describes a rectangular box with a width of 50 pixels and a height of 12 pixels. Note that you need five values for a box with four corners: The start value and the four destination values for the lines.

3.3.2.4 DrawBorder function

Now that we've taken stock of the situation, let's take advantage of Intuition's command for drawing borders. To do this you need a completely defined `Border` structure and a coordinate table:

```
/* 3.3.2.4.A. testborderstruct */
struct Border TestBorder =
   {
   50, 20,
   2, 0,
   JAM1,
   5,
   &TestValues,
   NULL
   };
```

Let's use the table printed above as the coordinates. In addition, you need the `RastPort` and the offsets for the function:

```
struct RastPort *MyWindowRastPort;
MyWindowRastPort = MyWindow->RPort;

DrawBorder(MyWindowRastPort, TestBorder, 10L, 10L);
```

Add the `Border` structure, the coordinate table, the `RastPort` and the function call to `CustomScrteen.C` program. After starting the modified program, a small box appears in the window. Here is the complete program:

```
/*************************************
 *                                   *
 * Program: DrawBorder.c             *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  Border box    *
 *                     in Window     *
 *                                   *
 *************************************/
```

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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;


struct NewWindow FirstNewWindow =
   {
   160, 50,              /* LeftEdge, TopEdge   */
   320, 150,             /* Width, Height       */
   0, 1,                 /* DetailPen, BlockPen */
   NULL,                 /* IDCMP Flags         */
   WINDOWDEPTH |         /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                 /* First Gadget        */
   NULL,                 /* CheckMark           */
   (UBYTE *)"System programing test",
   NULL,                 /* Screen              */
   NULL,                 /* BitMap              */
   100, 50,              /* Min Width, Height   */
   640, 200,             /* Max Width, Height   */
   WBENCHSCREEN,         /* Typ                 */
   };

SHORT TestValue[] =
   {
    0,  0,
   50,  0,
   50, 12,
    0, 12,
    0,  0
   };

struct Border TestBorder =
   {
   50, 20,
   2, 0,
   JAM1,
   5,
   TestValue,
   NULL
   };



main()
   {
   struct RastPort *MyWindowsRastPort;

   Open_All();
```

```
    MyWindowsRastPort = FirstWindow->RPort;

    DrawBorder(MyWindowsRastPort, &TestBorder, 10L, 10L);

    Delay(180L);

    Close_All();
    }


/*************************************
 *                                   *
 * Function: Library and Window open *
 * ================================= *
 *************************************/

Open_All()

    {
    void          *OpenLibrary();
    struct Window *OpenWindow();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
       {
       printf("Intuition Library not found!\n");
       Close_All();
       exit(FALSE);
       }

    if (!(FirstWindow = (struct Window *)
        OpenWindow(&FirstNewWindow)))
       {
       printf("Window will not open!\n");
       Close_All();
       exit(FALSE);
       }
    }


/*************************************
 *                                   *
 * Function: Close all               *
 * ================================= *
 *                                   *
 *************************************/

Close_All()

    {
    if (FirstWindow)     CloseWindow(FirstWindow);
    if (IntuitionBase)   CloseLibrary(IntuitionBase);
    }
```

Here is the format of the new function:

```
DrawBorder(RastPort, Border, LeftOffset, RightOffset);
   -108       A0       A1        D0          D1
```

Finally, here is an example which combines several `Border` structures. This is how programmers create multicolor boxes around system requesters.

```
/* 3.3.2.4.B.multiborders */
SHORT WhiteValues[] =
   {
    0, 0,
   50, 0,
   50,12,
    0,12,
    0, 0
   };

SHORT RedValues[] =
   {
    0, 0,
   54, 0,
   54,16,
    0,16,
    0, 0
   };

struct Border WhiteBorder =
   {
   20, 20,
   1, 0,
   JAM1,
   5,
   &WhiteValues,
   NULL
   };

struct Border RedBorder =
   {
   18, 18,
   2, 0,
   JAM1,
   5,
   &RedValues,
   &WhiteBorder
   };


DrawBorder(MyWindowRastPort, &RedBorder, 10L, 10L);
```

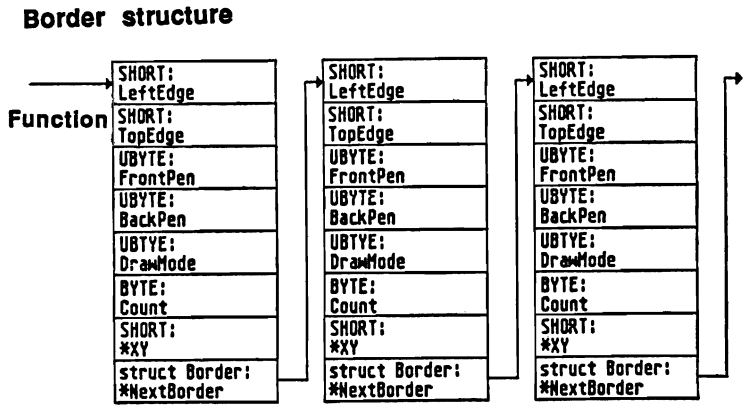


Figure 3.6

3.3.3 Graphic output

Now let's look at Intuition's most important output function: graphics. Look at the Workbench with its icons. Each icon and system gadget is a small graphic image.

3.3.3.1 Size, position, color, etc.

Like text and lines, each graphic also has a certain position in the window, which, as you know, acts as a relative offset to the coordinates in the function call.

In addition to the position specification, two values determine the size of the graphic. Each graphic can be viewed as a rectangular surface, for which you specify the width and height of the surface.

You may already be aware that each graphic on the Amiga is divided into several bit-planes. The bit-planes are basically individual sets of bits in memory, each controlling color output. One bitplane provides two colors, two bit-planes provide four colors, three provide eight colors, etc. We must specify the number of bit-planes used by our graphics so that in can be managed accordingly.

3.3.3.2 The Image structure

```
/* 3.3.3.2.A. imagestructure */
struct Image
{
0x00  00  SHORT LeftEdge;
0x02  02  SHORT TopEdge;
0x04  04  SHORT Width;
0x06  06  SHORT Height;
0x08  08  SHORT Depth;
0x0A  10  USHORT *ImageData;
0x0E  14  UBYTE PlanePick
0x0F  15  UBYTE PlaneOnOff;
0x10  16  struct Image *NextImage;
0x14  20
};
```

The first five values have been explained. Let's look at the remaining four.

`*ImageData` This pointer points to one or more data fields containing graphic data.

`*NextImage` This pointer lets you connect multiple `Image` structures. This allows you to easily generate complex graphics.

The Intuition `DrawImage()` function writes the graphic data into the bit-planes of our window or screen.

The `PlanePick` flag tells the function which planes of your graphic to transfer to those of the window. Each bit represents a plane:

Table 3.9: `PlanePick`, `PlaneOnOff`

PlanePick value (binary)	Planes used
00000000	None
00000001	Plane 0
00000010	Plane 1
00000100	Plane 2
	etc.
00000011	Plane 0 and 1
00000101	Plane 0 and 2
00000111	Plane 0, 1 and 2
	etc.

This table should make it easy to determine which bit-planes of the graphic are transferred to the window bit-planes. What can this method be used for? It can be used to turn off certain bit-planes of a graphic. This is often used to display a graphic in different colors.

The other flag, `PlaneOnOff`, makes use of the unused bit-planes. With the same values listed in the table above you can easily specify the bit-planes which are filled with ones. This gives you an easy way to fill a graphic with a color.

3.3.3.3 The graphic data in the Image structure

Let's take a look at the structure of graphic data. You can make graphic image development easier by drawing each item out on graph paper first. Each square represents a pixel. You must do this for all bit-planes, then you can calculate the values for your data.

First divide the rectangle into columns of four pixels each. You can view these as binary numbers, whereby each set pixel stands for a one, and calculate the value accordingly. Do this for each column in every line and write down the hexadecimal digits from left to right.

Here is a sample grid:

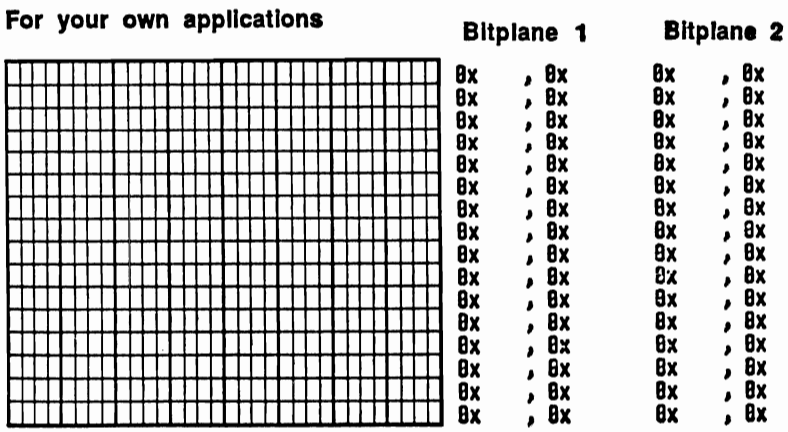


Figure 3.7

If your graphic consists of multiple bit-planes, you must list the data for the first bit-plane, then the data for the second, etc.

3.3.3.4 The `DrawImage` function

Now we come to some visible examples. Let's look at the first `Image` structure:

```
/* 3.3.3.4.A.imageexample */
struct ImageExample =
   {
   20, 10,
   16, 16,
   2,
   &ExampleData[0],
   3, 0,
   NULL
   };
```

This is a graphic representation of the `Image` structure chaining:

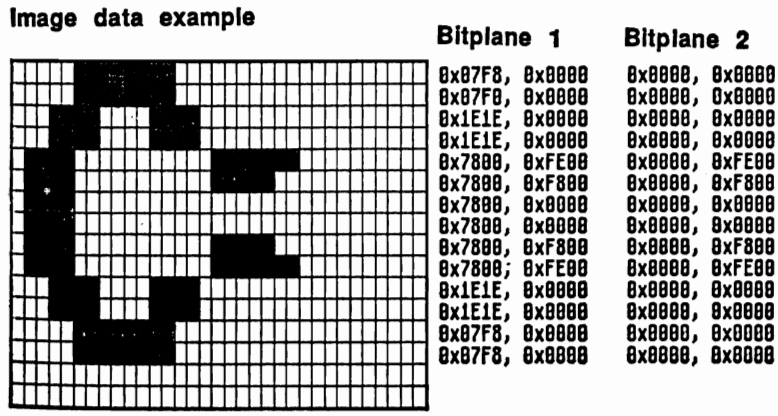


Figure 3.8

The data field for the example looks like this (remember when inserting this code in `CustomScreen1.c` that it must be defined before the `Image` structure):

```
/* 3.3.3.4.B. sampledata */
USHORT ExampleData[] =
   {
   /* first BitPlane */
   0x07F8, 0x0000,
   0x07F0, 0x0000,
   0x1E1E, 0x0000,
   0x1E1E, 0x0000,
   0x7800, 0xFE00,
   0x7800, 0xF800,
   0x7800, 0x0000,
   0x7800, 0x0000,
   0x7800, 0xF800,
   0x7800, 0xFE00,
   0x1E1E, 0x0000,
   0x1E1E, 0x0000,
   0x07F8, 0x0000,
   0x07F8, 0x0000,
   0x0000, 0x0000,
```

```
0x0000, 0x0000,
/* second BitPlane */
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0xFE00,
0x0000, 0xF800,
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0xF800,
0x0000, 0xFE00,
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0x0000,
0x0000, 0x0000
};
```

Finally, you have to find the `RastPort` of the window and call the `DrawImage()` function:

```
struct RastPort *MyWindowRastPort;
MyWindowRastPort = MyWindow->RPort;

DrawImage(MyWindowRastPort, Example, 10L, 10L);
```

The format:

```
DrawImage(RastPort, Image, LeftOffset, TopOffset);
   -114       A0        A1        D0          D1
```

Remember that your graphics chip can access only the chip memory (the lower 512K). There are two ways to ensure that graphic data are in this memory area. A linker option (+c in Aztec) transfers all data to the chip memory. This is probably the simplest method. But it can also be done within a program. To do this you must allocate space in chip memory and then copy the graphic data into it. Then you can initialize the graphic and display it. Here is the complete `DrawImage.c` program, remember to use the +C option in the Aztec linker to force the program into chip memory.

```
/*************************************
 *                                   *
 * Program: DrawImage.c              *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  Use +C linker *
 *                     option (Aztec)*
 *                                   *
 *************************************/
```

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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;


struct NewWindow FirstNewWindow =
   {
   160, 50,              /* LeftEdge, TopEdge    */
   320, 150,             /* Width, Height        */
   0, 1,                 /* DetailPen, BlockPen */
   NULL,                 /* IDCMP Flags          */
   WINDOWDEPTH |         /* Flags                */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                 /* First Gadget         */
   NULL,                 /* CheckMark            */
   (UBYTE *)"System programming test",
   NULL,                 /* Screen               */
   NULL,                 /* BitMap               */
   100, 50,              /* Min Width, Height    */
   640, 200,             /* Max Width, Height    */
   WBENCHSCREEN,         /* Typ                  */
   };

USHORT ExampleData[] =
   {
   /* first BitPlane */
   0x07F8, 0x0000,
   0x07F0, 0x0000,
   0x1E1E, 0x0000,
   0x1E1E, 0x0000,
   0x7800, 0xFE00,
   0x7800, 0xF800,
   0x7800, 0x0000,
   0x7800, 0x0000,
   0x7800, 0xF800,
   0x7800, 0xFE00,
   0x1E1E, 0x0000,
   0x1E1E, 0x0000,
   0x07F8, 0x0000,
   0x07F8, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000,
   /* second BitPlane */
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0x0000,
   0x0000, 0xFE00,
   0x0000, 0xF800,
```

```
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x0000, 0xF800,
    0x0000, 0xFE00,
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000
    };

struct Image Example =
    {
    20, 10,
    32, 16,
    2,
    &ExampleData[0],
    3, 0,
    NULL
    };



main()
    {
    struct RastPort *MyWindowsRastPort;

    Open_All();

    MyWindowsRastPort = FirstWindow->RPort;

    DrawImage(MyWindowsRastPort, &Example, 10L, 10L);

    Delay(180L);

    Close_All();
    }


/*************************************
 *                                   *
 * Function: Library and Window open *
 * ================================= *
 *                                   *
 *************************************/

Open_All()

    {
    void          *OpenLibrary();
    struct Window *OpenWindow();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
       {
```

```
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&FirstNewWindow)))
      {
      printf("Window will not open!\n");
      Close_All();
      exit(FALSE);
      }
   }


/*************************************
 *                                   *
 * Function: Close all               *
 * ================================= *
 *                                   *
 *************************************/

Close_All()

   {
   if (FirstWindow)      CloseWindow(FirstWindow);
   if (IntuitionBase)    CloseLibrary(IntuitionBase);
   }
```

3.3.4 Examples of graphic applications

We want to show you a few applications which you may find of interest.

You can see how easy it is to initialize the structure values. The large number of parameters allows you to configure graphics just the way you want them. This can be a problem in itself. If you have a number of identical strings, for example, it gets annoying if you have to constantly put the same values into the `IntuiText` structures. In this section we'll present a function which will help you out.

In addition, this section includes some small graphics, texts and lines which you will use in defining the gadgets described in Section 3.4. We're expanding the "toolkit" needed to write our C source code editor, a little at a time.

3.3.4.1 Easy text definition

Often you'll find that you have to use `IntuiText` structures to define multiple texts. Since you don't always choose a new character color and you usually limit yourselves to one font, defining twenty almost identical structures can be aggravating.

Here is a solution: First you define a default structure in which you enter all the parameters, but not the pointer to the string. Here is an example:

Structure 3.6: `DefaultText`

```
/*3.3.4.1.A.textdefinition */
struct TextAttr DefaultFont =
   {
   (STRPTR)"topaz.font",
   TOPAZ_EIGHTY,
   FS_NORMAL,
   FPF_ROMFONT
   };

struct IntuiText DefaultText =
   {
   1, 0,                    /* FrontPen, BackPen    */
   JAM2,                    /* DrawMode             */
   1, 1,                    /* LeftEdge, TopEdge    */
   &DefaultFont,            /* Font                 */
   NULL,                    /* Textpointer          */
   NULL                     /* NextText             */
   };
```

The `TextAttr` structure is also initialized here for correct font usage.

For the strings which are later inserted into this structure, you define a pointer array of type `char` so that you can access each string easily:

```
/*3.3.4.1.B.stringaccess */
char *Textarray[] =
   {
   "First text string",
   "here is the second",
   "and another",
   "and here we have the last and the longest!"
   };
```

Now you just need an array of `IntuiText` structures into which you will copy the default structure values and then add the string pointer.

```
struct IntuiText AllTexts[4];
```

Let's look at the subroutine which does all this:

Function 3.3: Make_Text

```
/*3.3.4.1.C.Make_Text function */
Make_Text(Number, Text, Struktur)

int               Number;
char             *Text[];
struct IntuiText  Struktur[];

   {
   int i;

   for (i=0; i<Num,ber; i++)
      {
      Struktur[i]          = DefaultText;
      Struktur[i].IText    = (UBYTE *) Text[i];
      Struktur[i].NextText = &Struktur[i+1];
      }
   Struktur[Number-1].NextText = NULL;
   }
```

How it works

The subroutine does its work in a small loop. Here the structure defined as an array element is first assigned the default structure values. Then you add the pointer to the first string and join this first `IntuiText` structure to the second. The loop then executes again for the next value. To prevent the last structure from pointing to a nonexistent structure, the extra pointer must be deleted again.

If you don't want to link the texts together, you can omit the two lines responsible for this. Here is a complete listing of a demo program using the `Make_Text()` function:

```
/*************************************
 *                                   *
 * Program: Make_Text.c              *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  Make text     *
 * JLD     01/16/1988  Structures    *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;


struct NewWindow FirstNewWindow =
   {
   160, 50,             /* LeftEdge, TopEdge   */
   400, 150,            /* Width, Height       */
```

```
   0, 1,                /* DetailPen, BlockPen */
   NULL,                /* IDCMP Flags         */
   WINDOWDEPTH |        /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                /* First Gadget        */
   NULL,                /* CheckMark           */
   (UBYTE *)"System programming test",
   NULL,                /* Screen              */
   NULL,                /* BitMap              */
   100, 50,             /* Min Width, Height   */
   640, 200,            /* Max Width, Height   */
   WBENCHSCREEN,        /* Typ                 */
   };

struct TextAttr DefaultFont =
   {
   (STRPTR)"topaz.font",
   TOPAZ_EIGHTY,
   FS_NORMAL,
   FPF_ROMFONT
   };

struct IntuiText DefaultText =
   {
   1, 0,                /* FrontPen, BackPen   */
   JAM2,                /* DrawMode            */
   1, 1,                /* LeftEdge, TopEdge   */
   &DefaultFont,        /* Font                */
   NULL,                /* Textpointer         */
   NULL                 /* NextText            */
   };

char *Textfield[] =
   {
   "First Text text",
   "here is the second",
   "and one more",
   "here is the last and the longest text!"
   };

struct IntuiText AllText[4];



main()
   {
   struct RastPort *MyWindowsRastPort;

   Open_All();
   Make_Text(4, Textfield, AllText);

   MyWindowsRastPort = FirstWindow->RPort;
```

```
   PrintIText(MyWindowsRastPort, &AllText[0], 10L, 10L);

   Delay(180L);

   Close_All();
   }


/*************************************
 *                                   *
 * Function: Library and Window open *
 * ================================= *
 *                                   *
 *************************************/

Open_All()

   {
   void          *OpenLibrary();
   struct Window *OpenWindow();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&FirstNewWindow)))
      {
      printf("Window will not open!\n");
      Close_All();
      exit(FALSE);
      }
   }


/*************************************
 *                                   *
 * Function: Close all               *
 * ================================= *
 *                                   *
 *************************************/

Close_All()

   {
   if (FirstWindow)     CloseWindow(FirstWindow);
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }


Make_Text(Number, Text, Struktur)
```

```
int               Number;
char             *Text[];
struct IntuiText  Struktur[];

   {
   int i;

   for (i=0; i<Number; i++)
      {
      Struktur[i]          = DefaultText;
      Struktur[i].IText    = (UBYTE *) Text[i];
      Struktur[i].TopEdge  = 20+i*10;
      Struktur[i].NextText = &Struktur[i+1];
      }
   Struktur[Number-1].NextText = NULL;
   }
```

3.3.4.2 Borders for gadgets

This subsection prepares you for the next section, where we'll look at gadgets. These gadgets use all three output elements which you've seen so far. A box often delimits the clickable area of a gadget, so we will develop a `Border` structure here which you can use later.

Let's take single and double boxes placed around gadgets as the first example. Double boxes are used in ***DataRetrieve*** and ***BeckerText***, for example, to designate which is the default gadget.

Here is the single border (for eight letters):

```
/* 3.3.4.2.A. singleborder */
SHORT SingleValues[] =
   {
    0, 0,
   66, 0,
   66,10,
    0,10,
    0, 0
   };
```

```
struct Border SingleBorder =
   {
   0, 0,
   1, 0,
   JAM1,
   5,
   &SingleValues,
   NULL
   };
```

and the double border;

```
/*3.3.4.2.B.doubleborder*/
SHORT DoubleValues[] =
   {
    0, 0,
   70, 0,
   70,14,
    0,14,
    0, 0
   };

struct Border DoubleBorder =
   {
   -2, -2,
   2, 0,
   JAM1,
   5,
   &DoubleValues,
   NULL
   };
```

The `DoubleBorder` call can be easily appended with the pointer:

```
SingleBorder.NexBorder = &DoubleBorder;
```

Next we offer a solution for the problem of defining a gadget border for text whose length is not known until the program is running. A `Border` structure and value array must be defined preceding the `main()` function. This would look like this:

```
/*3.3.4.2.C.borderlong*/
SHORT Values[] =
   {
    0, 0,
  999, 0,
  999,10,
    0,10,
    0, 0
   };

struct Border Bord =
   {
   0, 0,
   1, 0,
```

```
   JAM1,
   5,
   Values,
   NULL
   };
```

In addition, you need an `IntuiText` structure with everything but the pointer to the text:

```
/*3.3.4.2.D.generic_intuitext */
struct IntuiText Text =
   {
   1, 0,                    /* FrontPen, BackPen    */
   JAM2,                    /* DrawMode             */
   2, 2,                    /* LeftEdge, TopEdge    */
   NULL,                    /* Font (Standard)      */
   (UBYTE *) "Test",        /* Textpointer          */
   NULL                     /* NextText             */
   };
```

Both `Text` and `Bord` are written to `RastPort` at the same offsets:

```
DrawBorder(RastPort, Bord, 10L, 30L);
PrintIText(RastPort, Text, 10L, 30L);
```

Before this can be done, you have to call a function which calculates the correct border values for the text:

```
/*3.3.4.2.E.Calc*/
Calc_Border(Text)

struct IntuiText  Text;

   {
   int Width = 0;

   Width = IntuiTextLenght(Text);

   Values[2] = Width+4;
   Values[4] = Width+4;
   }
```

The only new thing is the `IntuiTextLength()` function, which returns the text width in pixels, while accounting for the font size. Once you have this value, you simply insert it in the array. Here is a complete example program:

```
/*************************************
 *                                   *
 * Program: Calc_Border.c            *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  for testing   *
```

```
 *                    only           *
 *                                   *
 ************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;


struct NewWindow FirstNewWindow =
   {
   160, 50,              /* LeftEdge, TopEdge   */
   320, 150,             /* Width, Height       */
   0, 1,                 /* DetailPen, BlockPen */
   NULL,                 /* IDCMP Flags         */
   WINDOWDEPTH |         /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                 /* First Gadget        */
   NULL,                 /* CheckMark           */
   (UBYTE *)"System programming test",
   NULL,                 /* Screen              */
   NULL,                 /* BitMap              */
   100, 50,              /* Min Width, Height   */
   640, 200,             /* Max Width, Height   */
   WBENCHSCREEN,         /* Typ                 */
   };

SHORT Values[] =
   {
    0, 0,
  999, 0,
  999,10,
    0,10,
    0, 0
   };

struct Border Bord =
   {
   0, 0,
   1, 0,
   JAM1,
   5,
   Values,
   NULL
   };

struct IntuiText Text =
   {
   1, 0,                 /* FrontPen, BackPen   */
```

```
    JAM2,                   /* DrawMode            */
    2, 2,                   /* LeftEdge, TopEdge   */
    NULL,                   /* Font (Standard)     */
    (UBYTE *) "Test",       /* Textpointer         */
    NULL                    /* NextText            */
    };

void Open_All(), Close_All();

main()
    {

    struct RastPort *RP;

    Open_All();

    RP = FirstWindow->RPort;

    Calc_Border(&Text);


    DrawBorder(RP, &Bord, 10L, 30L);
    PrintIText(RP, &Text, 10L, 30L);

    Delay(180L);

    Close_All();
    }
/*******************************
*                              *
* Function: Open All           *
*                              *
*******************************/


    void Open_All()
    {
    void          *OpenLibrary();
    struct Window *OpenWindow();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
       {
       printf("Intuition Library not found!\n");
       Close_All();
       exit(FALSE);
       }

    if (!(FirstWindow = (struct Window *)
        OpenWindow(&FirstNewWindow)))
       {
       printf("Window will not open!\n");
       Close_All();
       exit(FALSE);
       }
    }
```

```
/*************************************
 *                                   *
 * Function: Close All               *
 *                                   *
 *************************************/

   void Close_All()
   {
   if (FirstWindow)      CloseWindow(FirstWindow);
   if (IntuitionBase)    CloseLibrary(IntuitionBase);
   }


Calc_Border(Text)

struct IntuiText  *Text;

   {
   int Width = 0;

   Width = IntuiTextLength(Text);

   Values[2] = Width+4;
   Values[4] = Width+4;
   }
```

3.3.4.3 Symbols say more than words

As an extension to this section on images, we want to introduce a graphic which we could use later in our editor as a gadget icon.

It is somewhat annoying for the user if he always has to drag and resize the window to make it fill the screen just to click a gadget. It's just as annoying to reduce it in size.

Since you can do all this with window functions, you can easily perform this action using a gadget. Now you need an icon for the gadget.

The icon represents a small window and a large window, with a line connecting them to suggest the change in size.

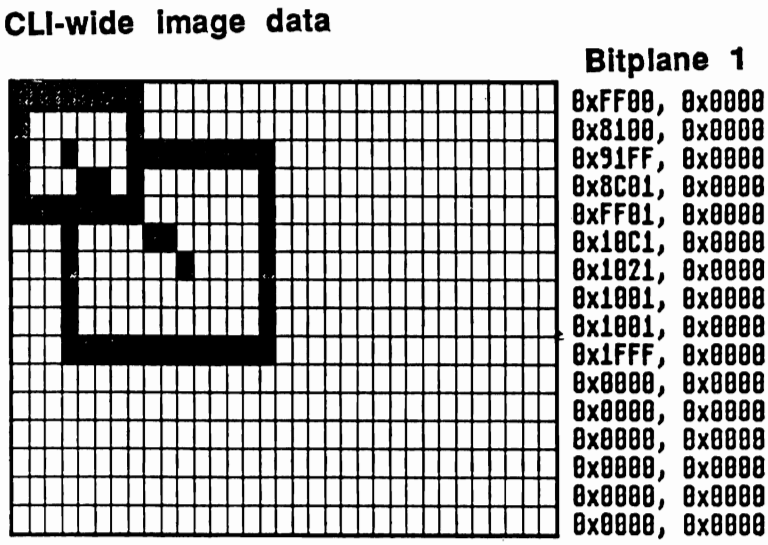


Figure 3.9
Resize Gadget

It shouldn't be a problem for you to build an `Image` structure from the picture and the graphic data.

3.4 Gadgets

Application programs need the input of general information, like text or data in order to operate properly. In addition, they need instructions to guide them in their work.

As an experienced Amiga owner you know that gadgets provide the easiest way to send information to the computer. These are rectangular selectable fields displayed in windows or screens. Programs react when you click the mouse button with the pointer on a gadget.

The Amiga can support more than just simple gadgets which just pass on a selection. Some gadgets can be turned off and on, some can be used for text input, and others can specify positions. These different gadget types can be further divided into subgroups. Using gadgets, you have many different ways to get information from the user.

You wouldn't be able to recognize these gadgets if they were only placed on the screen, Intuition allows the user to "decorate" these gadgets with borders, graphics and text. This is why the display techniques explained in the previous section are so important.

3.4.1 Different gadgets, different applications

Intuition offers two primitive gadget groups. The first group contains the system gadgets. Intuition offers direct support to access these gadges, and predefines their appearance and operating mode. See Section 3.1 for more information. The second group is intended for the programmer and supports custom gadgets. You can select from three different gadget types which have completely different properties.

Boolean gadgets

Boolean gadgets are intended for very simple commands and data which consist of just a boolean value (true or false). They offer two modes:

The first mode is called "hit select." This means that Intuition sends a report when you click the gadget with the left mouse button (called the select button in this case). The appearance of the gadget changes while the mouse selects it. If you move the mouse pointer away or release the select button, the appearance reverts to its original form. This mode can be used to issue very simple, specific commands.

With the second mode, "toggle select," you turn the gadget on with the first click. It then changes its appearance as if the mouse button were being held on it. A second click changes the gadget to its original state. This method is particularly good for turning choices on and off.

Proportional gadgets

If you want to represent areas or positions, this is the gadget type for you. With it you can display specific proportions in one or two dimensions and let the user of your program change them.

These gadgets consist of a rectangular box containing an object. We're using a generic term here because you can define any graphic you like as the object. For example, in Preferences the screen position object represents the edge of the Workbench screen, which you can set using a proportional gadget. In this case it is two-dimensional. An example of a one-dimensional proportional gadget is the scroll bar in a directory window. The bar, your "object," represents the size of the window display and can be moved back and forth.

For each proportional gadget you can define the size of the scroll box (container), the object used (knob), and the size of the object.

String gadgets

The string gadget allows simple text input. Like a proportional gadget, there is a "container" into which you enter a single line of text. The length is irrelevant: Intuition scrolls the text as necessary.

Editing is made easier by additional functions like `Undo`.

Now you know all of the gadget types and their basic properties. Let's look at the details of each type.

3.4.1.1 Boolean gadgets

One of the most common gadget types is the boolean gadget. Boolean gadgets appear three times in the system gadgets: Close gadget, front gadget and back gadget. Boolean gadgets represent the basic gadget type, which is why we will discuss them in such detail.

Each gadget can be assigned a position, height and width. Let's take a look at the structure before we go into the other values:

```
/* 3.4.1.1.A.booleanstruct */
struct Gadget
{
0x00  00  struct Gadget *NextGadget;
0x04  04  SHORT LeftEdge;
0x06  06  SHORT TopEdge;
0x08  08  SHORT Width;
0x0A  10  SHORT Height;
```

```
0x0C  12  USHORT Flags;
0x0E  14  USHORT Activation;
0x10  16  USHORT GadgetType;
0x12  18  APTR GadgetRender;
0x16  22  APTR SelectRender;
0x1A  26  struct IntuiText *GadgetText;
0x1E  30  LONG MutualExclude;
0x22  34  APTR SpecialInfo;
0x26  38  USHORT GadgetID;
0x28  40  APTR UserData;
0x2C  44
};
```

Just like all of the other Intuition structures you have looked at, gadgets can be linked together. This is the purpose of the first pointer. After the values for the left and top edges, height, and width you find a flag which performs several tasks.

Flags first show you the settings for the gadget graphic. You can specify how the appearance of the gadget is changed when it is clicked. There are four GADGHIGHBITS available for this:

GADGHCOMP — All points in the click area are complemented (reversed).

GADGHBOX — A box is drawn around the click area.

GADGHIMAGE — A different Image or Border is displayed.

GAGDHNONE — No change takes place. The flag also contains the position specifications for the gadget:

GRELBOTTOM — When you set this flag, the TopEdge variable of the structure describes an offset relative to the bottom of the window. If this flag isn't set, then TopEdge contains a relative specification from the upper border of the window.

GRELRIGHT — When you set this flag, the LeftEdge variable of the structure describes an offset relative to the right border of the window. If this flag isn't set, then LeftEdge is relative to the left border.

The size of the gadget can also be set in relationship to the window:

GRELWIDTH — If you set this flag, the value of the Width variable is subtracted from the window width and the result then represents the actual width of the gadget. If you do not set this flag, the Width represents an absolute value.

GRELHEIGHT If you set this flag, the value of the `Height` variable is subtracted from the window height and the result represents the actual height of the gadget. If you clear the flag, the value is interpreted as absolute.

There are three flags for gadget status:

GADGIMAGE If you set this flag, the `ImageRender` variable is assumed to point to an `Image` instead of a `Border`. The meaning also extends to the `SelectRender` variable.

GADGDISABLED
This flag is set (it can be set only at the beginning and thereafter it can only be read) when the gadget is unselectable.

SELECTED This flag is set (it can be set only at the beginning and thereafter it can only be read) when the gadget is selected.

Here are the flags together with their hex values:

Table 3.10: Gadget flags

Flag	Hex value
GADGHIGHBITS	0x0003L
GADGHCOMP	0x0000L
GADGHIMAGE	0x0002L
GADGHBOX	0x0001L
GADGHNONE	0x0003L
GRELBOTTOM	0x0008L
GRELRIGHT	0x0010L
GRELWIDTH	0x0020L
GRELHEIGHT	0x0040L
GADGIMAGE	0x0004L
SELECTED	0x0080L
GADGDISABLED	0x0100L

Now we come to the next value of the `Gadget` structure. This value is a set of flags which contain information about how the gadget is activated and whether it is located in the window border.

Let's first look at the flags which have to do with the activation status:

TOGGLESELECT
This flag defines a certain type of boolean gadget. Thereafter the gadget is treated as a switch: On the first click it turns on and it remains in this state until clicked again.

GADGIMMEDIATE
With simple boolean gadgets the programmer can choose between two select modes: If this mode is chosen, the program receives a message as soon as the gadget is clicked.

RELVERIFY If this flag is set, Intuition waits with the message until the user also releases the button in the click area. This allows the user to change his mind after having pressed the mouse button.

There are still more variations for the position of the gadget. These refer more to the management of the gadget position than the position of the gadget proper. You know that the border of a GIMMEZEROZERO window is handled specially so that the graphic in the window cannot be disturbed.

You can also place your custom gadgets in the window border. To do this, set the flags which follow.

These flags determine the gadget's location:

RIGHTBORDER The gadget is placed in the right window border.

LEFTBORDER The gadget is placed in the left border of the window.

TOPBORDER The gadget is placed in the top border of the window.

BOTTOMBORDER
The gadget is placed in the bottom border of the window.

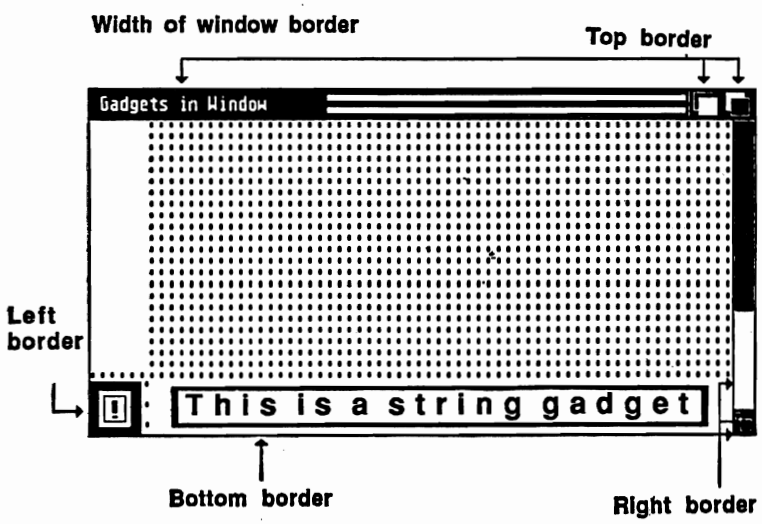


Figure 3.10

The following table lists all of the activation flags:

Table 3.11: Activation flags

Activation flag	Hex value	Comment
TOGGLESELECT	0x0100L	Switch gadget
RELVERIFY	0x0001L	
GADGIMMEDIATE	0x0002L	
RIGHTBORDER	0x0010L	Position in GZZ border
LEFTBORDER	0x0020L	
TOPBORDER	0x0040L	
BOTTOMBORDER	0x0080L	
STRINGCENTER	0x0200L	Reserved for string gadgets
STRINGRIGHT	0x0400L	
LONGINT	0x0800L	For integer gadgets
ALTKEYMAP	0x1000L	Custom keyboard table
BOOLEXTEND	0x2000L	
ENDGADGET	0x0004L	For requester
FOLLOWMOUSE	0x0008L	For IDCMP

Let's return to the `Gadget` structure. We now come to the last flag value which has to do with the type of gadget. `GadgetType` also contains more information:

`BOOLGADGET` A boolean gadget

`PROPGADGET` A proportional gadget (more on this later)

`STRGADGET` A string gadget with which simple text input can be performed. For an integer gadget, set the `LONGINT` flag in the `Flags` variable of the structure.

These three flags determine the gadget type. With two more flags, which are set by the system, you can tell where the gadget came from and where it lies.

`SCRGADGET` specifies that the gadget belongs to a screen and

`SYSGADGET` identifies the system gadgets in a list.

If a gadget is placed in the border of a `GZZ` window, the `GZZGADGET` flag must also be set.

Table 3.12 Gadget types

Gadget Type	Hex value	Comment
BOOLGADGET	0x0001L	
GADGET0002	0x0002L	Unused
PROPGADGET	0x0003L	
STRGADGET	0x0004L	
SYSGADGET	0x8000L	System gadget
SCRGADGET	0x4000L	Screen gadget
GZZGADGET	0x2000L	
GIMMEZEROZERO		Window gadget
REQGADGET	0x1000L	Requester gadget

Finally we come to the visible characteristics of the gadgets. We will concern ourselves with the graphic that normally associates with gadgets. Height, width and position are used to set the position and dimensions of the click field. Now it is time to make these visible to the user as well. To do this use a `Border` or `Image` structure. A pointer to one of these two structure types is stored in the `GadgetRender` variable. This graphic is then drawn when the gadget is displayed.

If you have also specified that a different appearance is desired when the gadget is clicked, then another pointer to an `Image` or `Border` structure must be in the `SelectRender` variable.

You can't just use a `Border` structure one time and an `Image` structure the next. As you can see above, a flag exists for each mode. This means that the `GAGDIMAGE` flag must be set for an `Image` structure and cleared for a `Border` structure.

`GADGIMAGE` also affects `SelectRender`, but it doesn't say that a pointer has to be stored there. Intuition looks for a structure pointer there only if the `GADGIMAGE` flag is set.

Intuition also offers a pointer to an `IntuiText` structure so that you can also put a label on your gadget. This text has nothing to do with selecting the gadget.

The last four values of the structure are for more complex applications. Since you are primarily interested in practical applications, lets ignore them for the moment. We will come back to them later in our discussion.

The first gadget

Now let's turn to programming. We have prepared the following gadget structure for the first example:

```
/* 3.4.1.1.B.boolgadget */
struct Gadget BoolGadget =
   {
   NULL,                    /* NextGadget          */
   10, 40,                  /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
```

```
   GADGHBOX,                  /* Flags              */
   RELVERIFY,                 /* Activation         */
   BOOLGADGET,                /* Gadget Type        */
   (APTR)&GadgetBorder,       /* GadgetRender       */
   NULL,                      /* Select Render      */
   &GadgetText,               /* GadgetText         */
   NULL,                      /* MutualExclude      */
   NULL,                      /* SpecialInfo        */
   1,                         /* GadgetID           */
   NULL,                      /* UserData           */
   };
```

Structure description

This defines a gadget of type `boolean`, which has a rectangular click field in the upper left corner of the window. When the gadget is selected, a border is drawn around the field. The gadget doesn't activate until the selection is verified. The gadget has a border and a label to make it recognizable and to explain its purpose to the user.

Something new is the value `GadgetID`, where you can enter a value from 0 to 65535. This is used to distinguish between gadgets. All you need to add are the `Border` structure, the coordinate table and the text. Here they are:

```
/* 3.4.1.1.C.gadgettext */
struct IntuiText GadgetText =
   {
   2, 0, JAM2, 4, 7, NULL, (UBYTE *)"Gadget" ,NULL,
   };

SHORT GadgetPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
   };

struct Border GadgetBorder =
   {
   -1, -1, 1, 0, JAM1, 5, GadgetPairs, NULL,
   };
```

Insert the new program fragments into `First_window.c` program from Section 3.1. Remember that the `IntuiText` and `Border` structures must be defined preceding the `Gadget` structure. Finally, add a pointer to the window structure which points to a user gadget. The entire window structure then looks like this:

```
/* 3.4.1.1.D.firstnewwindow */
struct NewWindow FirstNewWindow =
   {
   160, 50,                   /* LeftEdge, TopEdge   */
   320, 150,                  /* Width, Height       */
   0, 1,                      /* DetailPen, BlockPen */
   CLOSEWINDOW |              /* IDCMP Flags         */
   GADGETUP,
   WINDOWDEPTH |              /* Flags               */
```

```
WINDOWSIZING |
WINDOWDRAG | WINDOWCLOSE |
SMART_REFRESH,
&BoolGadget,              /* First Gadget        */
NULL,                     /* CheckMark           */
(UBYTE *)"Gadget Programming Test",
NULL,                     /* Screen              */
NULL,                     /* BitMap              */
100, 50,                  /* Min Width, Height   */
640, 200,                 /* Max Width, Height   */
WBENCHSCREEN,             /* Type                */
};
```

Take a look at the new IDCMP flag, which sends you messages whenever a gadget is selected.

With this addition you also have to extend the testing of the IDCMP flag for this new case. You can get messages from a self-defined user gadget as well as from the close gadget:

```
/*3.4.1.1.E.othergadget_loop */
   FOREVER
      {
      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
         Wait(1L << FirstWindow->UserPort->mp_SigBit);
         continue;
         }
      MessageClass = message->Class;
      code = message->Code;
      ReplyMsg(message);
      switch (MessageClass)
         {
         case GADGETUP    : nr = nr + 1;
              printf("Gadget activated %d times!\n", nr);
                            break;

         case CLOSEWINDOW : Close_All();
                            exit(TRUE);
                            break;

         }
      }
```

The test loop here offers another important advantage over the previous one: The Wait() function places the task in the wait state until a signal occurs. Therefore the loop doesn't use any processor time when it has nothing to do.

The test for the new gadget is easy. The switch() statement tests to see if a user-defined gadget has been selected. Since you only used one, you don't have to sort things out any further, and output a counter

which indicates how many times the gadget has been clicked. It's important not to forget to declare this variable.

How it works After the program starts, a window containing a boolean gadget appears on the Workbench screen. The program receives a message whenever you click on this gadget with the mouse button. Since you have set RELVERIFY, the button must also be released while the pointer is within the click field. A message with the number of clicks appears in the DOS window. You can end the program by clicking the close gadget.

Boolean gadgets always work well for simple processes. They allow the user to easily issue or confirm a command, or terminate an action. Here is a complete example listing for a boolean gadget. Remember to specify the 32-bit version of the libraries if you use the +L option of the Aztec compiler.

```
/*************************************
 *                                   *
 * Gadgets : Boolean-Gadget.c        *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  only for      *
 *                     testing       *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


SHORT GadgetPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
   };

struct Border GadgetBorder =
   {
   -1, -1, 1, 0, JAM1, 5, GadgetPairs, NULL,
   };

struct IntuiText GadgetText =
   {
   2, 0, JAM2, 4, 7, NULL, (UBYTE *)"Gadget" ,NULL,
   };
```

```
struct Gadget BoolGadget =
   {
   NULL,                  /* NextGadget          */
   10, 40,                /* LeftEdge, TopEdge   */
   60, 20,                /* Width, Height       */
   GADGHBOX,              /* Flags               */
   RELVERIFY,             /* Activation          */
   BOOLGADGET,            /* Gadget Type         */
   (APTR)&GadgetBorder,   /* GadgetRender        */
   NULL,                  /* Select Render       */
   &GadgetText,           /* GadgetText          */
   NULL,                  /* MutualExclude       */
   NULL,                  /* SpecialInfo         */
   1,                     /* GadgetID            */
   NULL,                  /* UserData            */
   };

struct NewWindow FirstNewWindow =
   {
   160, 50,               /* LeftEdge, TopEdge   */
   320, 150,              /* Width, Height       */
   0, 1,                  /* DetailPen, BlockPen */
   CLOSEWINDOW |          /* IDCMP Flags         */
   GADGETUP,
   WINDOWDEPTH |          /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,

   &BoolGadget,           /* First Gadget        */
   NULL,                  /* CheckMark           */

   (UBYTE *)"System programming test",

   NULL,                  /* Screen              */
   NULL,                  /* BitMap              */

   100, 50,               /* Min Width, Height   */
   640, 200,              /* Max Width, Height   */

   WBENCHSCREEN,          /* Type                */
   };


main()
   {
   ULONG MessageClass;
   USHORT code, nr = 0;


   struct Message *GetMsg();

   Open_All();
```

```
    RefreshGadgets(&BoolGadget, FirstWindow, NULL);


    FOREVER
        {
        if ((message = (struct IntuiMessage *)
            GetMsg(FirstWindow->UserPort)) == NULL)
            {
            Wait(1L << FirstWindow->UserPort->mp_SigBit);
            continue;
            }
        MessageClass = message->Class;
        code = message->Code;
        ReplyMsg(message);
        switch (MessageClass)
            {
            case GADGETUP      : nr += 1 ;
                 printf("Gadget activated %u times!\n", nr);
                                 break;

            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;
            }
        }
    }


Open_All()
    {
    void *OpenLibrary();
    struct Window *OpenWindow();

    if (!(IntuitionBase = (struct IntuitionBase *)
          OpenLibrary("intuition.library", 0L)))
        {
        printf("Intuition Library not found!\n");
        Close_All();
        exit(FALSE);
        }

    if (!(FirstWindow = (struct Window
*)OpenWindow(&FirstNewWindow)))
        {
        printf("Window will not open!\n");
        Close_All();
        exit(FALSE);
        }
    }

Close_All()
    {
    if (FirstWindow)      CloseWindow(FirstWindow);
    if (IntuitionBase)    CloseLibrary(IntuitionBase);
    }
```

3.4.1.2 Proportional gadgets

All other gadget types are extensions of the boolean gadget, including the proportional gadget. This gadget is used for the Workbench scroll bars. This is useful when the material to be displayed is larger than the actual display area. Then a scroll bar indicates the relative size of the display area and allows the user to view a different portion of the whole. Preferences uses a two-dimensional proportional gadget to allow you to set the position of the Workbench screen. This is another application of this gadget type.

For the first proportional gadget, let's take a simple basic gadget structure with a long vertical click field:

```
/* 3.4.1.2.A.propgadget */struct Gadget PropGadget =
   {
   NULL,                    /* NextGadget          */
   100, 20,                 /* LeftEdge, TopEdge   */
   20, 60,                  /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   PROPGADGET,              /* Gadget Type         */
   (APTR)Buffer,            /* GadgetRender        */
   NULL,                    /* Select Render       */
   NULL,                    /* GadgetText          */
   NULL,                    /* MutualExclude       */
   &ExampleProp,            /* SpecialInfo         */
   2,                       /* GadgetID            */
   NULL,                    /* UserData            */
   };
```

As you can see, this structure is a little short on arguments. But you don't need many: You don't need a text in this example, and a border around the container is drawn automatically for proportional gadgets.

We have, however, defined the value `SpecialInfo`. This pointer is used whenever a boolean gadget isn't enough. It then points to an extension structure. In this case you need a `PropInfo` structure, which looks as follows:

```
/* 3.4.1.2.B.propinfostruct*/
struct PropInfo
{
0x00  00  USHORT Flags;
0x02  02  USHORT HorizPot;
0x04  04  USHORT VertPot;
0x06  06  USHORT HorizBody;
0x08  08  USHORT VertBody;
0x0A  10  USHORT CWidth;
0x0C  12  USHORT CHeight;
```

```
0x0E  14  USHORT HPotRes;
0x10  16  USHORT VPotRes;
0x12  18  USHORT LeftBorder;
0x14  20  USHORT TopBorder;
0x16  22
};
```

So you see, you need more data for a proportional gadget. These contain the dimensions of the container and the knob. The knob is the actual graphic that is moved by the user. The container is the actual area in which the knob can be moved. Here are the descriptions of each parameter:

`Flags` You need a few additional values for proportional gadgets. You can select from among the following arguments:

`AUTOKNOB` This flag indicates that you have not defined a custom graphic for the knob and that Intuition should use a rectangle as default. Intuition creates this rectangle, and automatically adapts it to any situation. The size of the knob can be set with the next two structure values.

`FREEHORIZ` If you set this flag, the knob can be moved horizontally. The knob can be moved in both directions if you also set the `FREEVERT` flag.

`FREEVERT` If you set this flag, the knob can be moved vertically. The knob can be moved in both directions if you also set the `FREEHORIZ` flag.

`KNOBHIT` You cannot set this flag yourself. Intuition sets this flag if the knob was just clicked with the mouse.

`PROPBORDERLESS` Normally Intuition draws a border around the gadget box. Setting this flag suppresses the border.

`HorizPot` Specifies the horizontal position of the knob.

`VertPot` Specifies the vertical position of the knob.

`HorizBody` Here you can set the horizontal increment of knob movement. This is the value which the knob moves when you click in the container, but not on the knob itself.

`VertBody` Here you can set the vertical increment of knob movement. This is the value which the knob moves when you click in the container, but not on the knob itself.

None of the following values need be set by the programmer. They are calculated by Intuition and can be read after the gadget has been initialized.

`CWidth`	Container width
`CHeight`	Container height
`HPotRes`	Horizontal knob resolution
`VPotRes`	Vertical knob resolution
`LeftBorder`	Current position of the left border of the container
`TopBorder`	Current position of the top border of the container

The example code used to access the `PropInfo` structure would look something like this:

```
/*3.4.1.2.C.exampleprop*/
struct PropInfo ExampleProp =
{
   AUTOKNOB | FREEVERT,      /*    Flags          */
   0x8000,                   /*    HorizPot       */
   0x8000,                   /*    VertPot        */
   0x0800,                   /*    HorizBody      */
   0x0800,                   /*    VertBody       */
   0,                        /*    CWidth         */
   0,                        /*    CHeight        */
   0,                        /*    HPotRes        */
   0,                        /*    VPotRes        */
   0,                        /*    LeftBorder     */
   0                         /*    TopBorder      */
};
```

Now you can insert the structure definitions in your program. Remember that the `PropInfo` structure must be defined first so that the value of `&ExampleProp` is known to the gadget structure. The same applies to the window structure, in which the address of the gadget structure is set in place.

Even after you've done all this, you're still not finished. With a proportional gadget you get a message when the gadget has been released (`GADGETUP`) instead of when you "press it" (`GADGETDOWN`). You have to specify this in your loop:

Program 3.3: Test loop for proportional gadget

```
/*3.4.1.2.D.prop forever */
   FOREVER
      {

      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
```

```
            Wait(1L << FirstWindow->UserPort->mp_SigBit);
            continue;
            }
         MessageClass = message->Class;
         code = message->Code;
         ReplyMsg(message);
         switch (MessageClass)
            {
            case GADGETUP    : printf("Position: u%\n",
                                      ExampleProp.VertPot);
                               break;

            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;

            case GADGETDOWN  : nr += 1;
                               printf("Gadget activated for
                                     the %nr time!\n", nr);
                               break;

            }
         }
```

Here is a complete program listing for creating a proportional gadget:

```
/*************************************
 *                                   *
 * Program: Proportional-Gadget.c    *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  for testing   *
 *                     only          *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;

WORD Buffer[4];

struct PropInfo ExampleProp =
{
   AUTOKNOB | FREEVERT, /*    Flags          */
   0x8000,              /*    HorizPot       */
   0x8000,              /*    VertPot        */
   0x0800,              /*    HorizBody      */
   0x0800,              /*    VertBody       */
```

```
    0,                      /*    CWidth          */
    0,                      /*    CHeight         */
    0,                      /*    HPotRes         */
    0,                      /*    VPotRes         */
    0,                      /*    LeftBorder      */
    0                       /*    TopBorder       */
};

struct Gadget PropGadget =
    {
    NULL,                   /* NextGadget         */
    100, 20,                /* LeftEdge, TopEdge  */
    20, 60,                 /* Width, Height      */
    GADGHCOMP,              /* Flags              */
    RELVERIFY,              /* Activation         */
    PROPGADGET,             /* Gadget Type        */
    (APTR)Buffer,           /* GadgetRender       */
    NULL,                   /* Select Render      */
    NULL,                   /* GadgetText         */
    NULL,                   /* MutualExclude      */
    (APTR)&ExampleProp,  /* SpecialInfo        */
    2,                      /* GadgetID           */
    NULL,                   /* UserData           */
    };

struct NewWindow FirstNewWindow =
    {
    160, 50,                /* LeftEdge, TopEdge  */
    320, 150,               /* Width, Height      */
    0, 1,                   /* DetailPen, BlockPen */
    CLOSEWINDOW |           /* IDCMP Flags        */
    GADGETUP,
    WINDOWDEPTH |           /* Flags              */
    WINDOWSIZING |
    WINDOWDRAG |
    WINDOWCLOSE |
    SMART_REFRESH,
    &PropGadget,            /* First Gadget       */
    NULL,                   /* CheckMark          */
    (UBYTE *)"Gadget Programmng Test",
    NULL,                   /* Screen             */
    NULL,                   /* BitMap             */
    100, 50,                /* Min Width, Height  */
    640, 200,               /* Max Width, Height  */
    WBENCHSCREEN,           /* Type               */
    };



main()
    {
    ULONG MessageClass;
    USHORT code, nr = 0;

    struct Message *GetMsg();
```

```
   Open_All();

   FOREVER
      {
      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
         Wait(1L << FirstWindow->UserPort->mp_SigBit);
         continue;
         }
      MessageClass = message->Class;
      code = message->Code;
      ReplyMsg(message);
      switch (MessageClass)
         {
         case GADGETUP    : printf("Position: %u\n",
ExampleProp.VertPot);
                            break;

         case CLOSEWINDOW : Close_All();
                            exit(TRUE);
                            break;

         case GADGETDOWN  : nr += 1;
                            printf("Gadget activated %d.
times!\n", nr);
                            break;

         }
      }
   }


/*************************************
 *                                   *
 * Function: Library and Window open *
 *                                   *
 *************************************/

Open_All()

   {
   void          *OpenLibrary();
   struct Window *OpenWindow();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&FirstNewWindow)))
```

```
    {
    printf("Window will not open!\n");
    Close_All();
    exit(FALSE);
    }
}

/*************************************
 *                                   *
 * Function: Close all               *
 *                                   *
 *************************************/

Close_All()

{
if (FirstWindow)      CloseWindow(FirstWindow);
if (IntuitionBase)    CloseLibrary(IntuitionBase);
}
```

3.4.1.3 The string gadget

String gadgets, like proportional gadgets, are an extension of boolean gadgets. As you might imagine, you insert a `StringInfo` structure instead of a `PropInfo` structure. This is about the same length as the `PropInfo` structure, and contains all of the extra information that a string gadget needs.

You have no doubt used a string gadget before. Even the Workbench offers them. When you change the name of a disk or file you select the `Rename` option in the Workbench menu. A small window appears into which you can enter the new name. This is a string gadget. In this case, the window is the same size as the gadget, but this is not important.

You have to set up a structure for a string gadget just like any other gadget. You must define a horizontal click field, since text input is limited to a single line.

```
/* 3.4.1.3.A. stringgadget */
struct Gadget StringGadget =
   {
   NULL,                    /* NextGadget          */
   50, 40,                  /* LeftEdge, TopEdge   */
   120, 10,                 /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY |              /* Activation          */
   STRINGCENTER,
   STRGADGET,               /* Gadget Type         */
```

```
(APTR)&GadgetBorder,     /* GadgetRender        */
NULL,                    /* Select Render       */
&GadgetText,             /* GadgetText          */
NULL,                    /* MutualExclude       */
(APTR)&StringInfo,       /* SpecialInfo         */
3,                       /* GadgetID            */
NULL,                    /* UserData            */
};
```

Some variables have additional or different values for a string gadget. The first is the `Flags` flag. It now describes the appearance of the cursor and not the entire click field.

Four more flags can be used in string gadget activation:

`STRINGCENTER`
: This flag centers the text in the gadget when set. The default setting (unset `STRINGCENTER` flag) left-justifies the text in the gadget.

`STRINGRIGHT`
: This flag right-justifies the text in the gadget when set. The default setting (unset `STRINGRIGHT` flag) left-justifies the text in the gadget.

`LONGINT`
: This flag indicates an integer gadget (lets the user enter numbers only).

`ALTKEYMAP`
: Setting this flag defines the value `AltKeyMap` in the `StringInfo` structure, thus allowing access to an alternate keyboard driver.

After you specify the gadget type with `STRGADGET`, you must not forget to identify the click field with a graphic. Borders are drawn automatically only for proportional gadgets. We used a simple `Border` structure here:

```
/*3.4.1.3.B.gadgetpairs*/
SHORT GadgetPairs[] =
   {
   0, 0, 122, 0, 122, 12, 0, 12, 0, 0,
   };

struct Border GadgetBorder =
   {
   -2, -2, 1, 0, JAM1, 5, GadgetPairs, NULL,
   };
```

The gadget text is not required, but you should include it so that the user knows what to enter:

```
/*3.4.1.3.C/gadgettext*/
struct IntuiText GadgetText =
   {
   2, 0, JAM2, -40, 1, NULL, (UBYTE *)"Text" ,NULL,
   };
```

The last additional information worth mentioning is the pointer to the `StringInfo` structure. This new structure contains the following parameters:

```
/* 3.4.1.3.D.stringinfostruct*/
struct StringInfo
{
0x00  00  UBYTE *Buffer;
0x04  04  UBYTE *UndoBuffer;
0x08  08  SHORT BufferPos;
0x0A  10  SHORT MaxChars;
0x0C  12  SHORT DispPos;
0x0E  14  SHORT UndoPos;
0x10  16  SHORT NumChars;
0x12  18  SHORT DispCount;
0x14  20  SHORT CLeft;
0x16  22  SHORT CTop;
0x18  24  struct Layer *LayerPtr;
0x1C  28  LONG LongInt;
0x20  32  struct KeyMap *AltKeyMap;
0x24  36
};
```

The first two values are the most important. They point to a buffer in which the strings are stored. The first buffer contains the text which was actually entered. The second buffer is used for the `Undo` editing feature. The user just has to press a key to restore the text.

You can quickly create such a buffer. You simply define an array with the corresponding number of characters. It looks like this:

```
#define STRINGSIZE 80
unsigned char StringBuffer[STRINGSIZE] = "Hello Amiga!";
unsigned char UndoBuffer  [STRINGSIZE];
```

You don't have to provide an undo buffer, but then the user can't use this function.

You can use `BufferPos` to initialize the cursor at a specific location. This is used when text is already in the buffer before the call, as in the example above.

Since Intuition does not know how large the text buffer is, you have to tell it in `MaxChars` how many characters can be entered. Note that the terminating null byte must be included in the string length.

The last value which can be set by the programmer is `DispPos`. The number which Intuition finds there designates the first character which is displayed in the gadget field.

All other values are handled by Intuition itself. You can read them for information.

`UndoPos`	Cursor position in the undo buffer.
`NumChars`	Number of characters currently in the buffer.
`DispCount`	Number of characters which can be displayed in the gadget field.
`CLeft`	The offset of the container from the left border of the window.
`CTop`	The offset of the container from the upper border.
`LayerPtr`	A pointer to the layer in which the gadget is located.
`LongInt`	After pressing <Return>, this variable contains the number entered for an integer gadget for later reading and processing.
`AltKeyMap`	As mentioned above, you can place a pointer to a custom keyboard table here which is then used for input.

Let's look at some practical applications now. Insert the gadget, `StringInfo`, and other structures into your standard program 3.1.

The definitions look like this:

```
/* 3.4.1.3.E.stringadditions */
#define STRINGSIZE 80
unsigned char StringBuffer[STRINGSIZE] = "Hello Amiga!";
unsigned char UndoBuffer  [STRINGSIZE];

struct StringInfo StringInfo =
   {
   &StringBuffer[0],        /* Buffer              */
   &UndoBuffer[0],          /* Undo Buffer         */
   0,                       /* Buffer Position     */
   STRINGSIZE,              /* MaxChars            */
   0,                       /* Display Position    */
   0,                       /* Undo Position       */
   0,                       /* NumChars            */
   0,                       /* Display Counter     */
   0, 0,                    /* CLeft, CTop         */
   NULL,                    /* LayerPtr            */
   0,                       /* LongInt             */
   NULL,                    /* AltKeyMap           */
   };
```

```
SHORT GadgetPairs[] =
   {
   0, 0, 122, 0, 122, 12, 0, 12, 0, 0,
   };

struct Border GadgetBorder =
   {
   -2, -2, 1, 0, JAM1, 5, GadgetPairs, NULL,
   };

struct IntuiText GadgetText =
   {
   2, 0, JAM2, -40, 1, NULL, (UBYTE *)"Text" ,NULL,
   };

struct Gadget StringGadget =
   {
   NULL,                        /* NextGadget          */
   50, 40,                      /* LeftEdge, TopEdge   */
   120, 10,                     /* Width, Height       */
   GADGHCOMP,                   /* Flags               */
   RELVERIFY |                  /* Activation          */
   STRINGCENTER,
   STRGADGET,                   /* Gadget Type         */
   (APTR)&GadgetBorder,         /* GadgetRender        */
   NULL,                        /* Select Render       */
   &GadgetText,                 /* GadgetText          */
   NULL,                        /* MutualExclude       */
   (APTR)&StringInfo,           /* SpecialInfo         */
   1,                           /* GadgetID            */
   NULL,                        /* UserData            */
   };

struct NewWindow FirstNewWindow =
   {
   160, 50,                     /* LeftEdge, TopEdge   */
   320, 150,                    /* Width, Height       */
   0, 1,                        /* DetailPen, BlockPen */
   CLOSEWINDOW |                /* IDCMP Flags         */
   GADGETUP,
   WINDOWDEPTH |                /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   &StringGadget,               /* First Gadget        */
   NULL,                        /* CheckMark           */
   (UBYTE *)"String Gadget Test",
   NULL,                        /* Screen              */
   NULL,                        /* BitMap              */
   100, 50,                     /* Min Width, Height   */
   640, 200,                    /* Max Width, Height   */
   WBENCHSCREEN,                /* Type                */
   };
```

For the program use a slightly modified test loop:

```
3.4.1.3.F.testloop */
      switch (MessageClass)
         {
         case GADGETUP    : printf("The string is: %s\n",

&StringBuffer[0]);
                            break;

         case CLOSEWINDOW : Close_All();
                            exit(TRUE);
                            break;
```

How it works A string gadget appears in the window. Clicking on the gadget allows you to edit the text in it. When you press the <Return> key, the program prints the string entered in the CLI window.

The program receives a GADGETUP message each time the user presses <Return> while in a string gadget. This can be used as a message that the input is finished. Here is a complete example program using a string gadget:

```
/*************************************
 *                                   *
 * Program : String-Gadget.c         *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  for testing   *
 *                     only          *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


#define STRINGSIZE 80
unsigned char StringBuffer[STRINGSIZE] = "Hello Amiga";
unsigned char UndoBuffer  [STRINGSIZE];

struct StringInfo StringInfo =
   {
   StringBuffer,        /* Buffer              */
   UndoBuffer,          /* Undo Buffer         */
   0,                   /* Buffer Position     */
   STRINGSIZE,          /* MaxChars            */
   0,                   /* Display Positoin    */
```

```
    0,                  /* Undo Position        */
    0,                  /* NumChars             */
    0,                  /* Display Counter      */
    0, 0,               /* CLeft, CTop          */
    NULL,               /* LayerPtr             */
    0,                  /* LongInt              */
    NULL,               /* AltKeyMap            */
    };

SHORT GadgetPairs[] =
    {
    0, 0, 122, 0, 122, 12, 0, 12, 0, 0,
    };

struct Border GadgetBorder =
    {
    -2, -2, 1, 0, JAM1, 5, GadgetPairs, NULL,
    };

struct IntuiText GadgetText =
    {
    2, 0, JAM2, -40, 1, NULL, (UBYTE *)"Text" ,NULL,
    };

struct Gadget StringGadget =
    {
    NULL,               /* NextGadget           */
    50, 40,             /* LeftEdge, TopEdge    */
    120, 10,            /* Width, Height        */
    GADGHCOMP,          /* Flags                */
    RELVERIFY |         /* Activation           */
    STRINGCENTER,
    STRGADGET,          /* Gadget Type          */
    (APTR)&GadgetBorder, /* GadgetRender        */
    NULL,               /* Select Render        */
    &GadgetText,        /* GadgetText           */
    NULL,               /* MutualExclude        */
    (APTR)&StringInfo,  /* SpecialInfo          */
    1,                  /* GadgetID             */
    NULL,               /* UserData             */
    };

struct NewWindow FirstNewWindow =
    {
    160, 50,            /* LeftEdge, TopEdge    */
    320, 150,           /* Width, Height        */
    0, 1,               /* DetailPen, BlockPen  */
    CLOSEWINDOW |       /* IDCMP Flags          */
    GADGETUP |
    GADGETDOWN,

    WINDOWDEPTH |       /* Flags                */
    WINDOWSIZING |
    WINDOWDRAG |
    WINDOWCLOSE |
    SMART_REFRESH,
```

```
    &StringGadget,          /* First Gadget          */
    NULL,                   /* CheckMark             */

    (UBYTE *)"System programming test",

    NULL,                   /* Screen                */
    NULL,                   /* BitMap                */

    100, 50,                /* Min Width, Height     */
    640, 200,               /* Max Width, Height     */

    WBENCHSCREEN,           /* Type                  */
    };


main()
    {
    ULONG MessageClass;
    USHORT code;

    struct Message *GetMsg();

    Open_All();

    RefreshGadgets(&StringGadget, FirstWindow, NULL);


    FOREVER
        {
        if ((message = (struct IntuiMessage *)
             GetMsg(FirstWindow->UserPort)) == NULL)
            {
            Wait(1L << FirstWindow->UserPort->mp_SigBit);
            continue;
            }
        MessageClass = message->Class;
        code = message->Code;
        ReplyMsg(message);
        switch (MessageClass)
            {
            case GADGETUP    : printf("The string is:
%s\n",&StringBuffer[0]);

            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;
            }
        }
    }


Open_All()
    {
    void *OpenLibrary();
    struct Window *OpenWindow();
```

```
   if (!(IntuitionBase = (struct IntuitionBase *)
         OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window *)
OpenWindow(&FirstNewWindow)))
      {
      printf("Window will not open!\n");
      Close_All();
      exit(FALSE);
      }
   }

Close_All()
   {
   if (FirstWindow)      CloseWindow(FirstWindow);
   if (IntuitionBase)    CloseLibrary(IntuitionBase);
   }
```

3.4.2 Gadgets in action

Now we will present an example of multiple applications of the three gadgets just described.

Gadgets are user friendly because of their graphic nature. But string and proportional gadgets are not without problems. That is why we want to employ a few examples that can be built into other programs, but have been designed specifically with our editor in mind.

Until now the gadgets were always connected with the `NewWindow` structure, and the program simply displayed them. This must be changed, because not all gadgets are needed when you first open a window. Gadgets are usually inserted temporarily into a window then removed from the window after the user response has been determined. The Intuition gadget functions are used to control the display of the gadgets.

The largest use of gadgets are in requesters. They are collections of multiple gadgets. Because we need many gadgets, we want to give you some examples that will be completed in the chapter on requesters.

3.4.2.1 Gadget functions

First we want to concern ourselves with the problem mentioned above, that not all gadgets are always wanted when opening a window. In many programs, having a gadget inserted in a window can be useful. Intuition has many functions that can do this.

Let's imagine that we've opened a normal window. Then, during program execution, a gadget is placed in the window structure and then appears on the screen.

We described the operation in single steps because we must proceed in the same single steps when programming. As a base program you will use the `First_window.c` program from section 3.1, including the global `NewWindow` structure. Define a toggle select boolean gadget with the following gadget structure, but don't place it in the `NewWindow` structure:

```
/*3.4.2.1.A.togglegadget */
SHORT GadgetPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0
   };

struct Border GadgetBorder =
   {
   -1, -1, 1, 0, JAM1, 5, GadgetPairs, NULL
   };

struct IntuiText ToggleText =
   {
   3, 0, JAM2, 4, 7, NULL, (UBYTE *)"Toggle" ,NULL
   };

struct Gadget ToggleGadget =
   {
   NULL,                    /* NextGadget          */
   120, 40,                 /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
   GADGHBOX,                /* Flags               */
   RELVERIFY |              /* Activation          */
   TOGGLESELECT,
   BOOLGADGET,              /* Gadget Type         */
   (APTR)&GadgetBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &ToggleText,             /* GadgetText          */
   NULL,                    /* MutualExclude       */
   NULL,                    /* SpecialInfo         */
   1,                       /* GadgetID            */
   NULL                     /* UserData            */
   };
```

A toggle gadget goes into the "selected" status when clicked on, and when they are clicked on again they are "unselected."

We place this gadget in the window using an Intuition function. It is called AddGadget() and has the following syntax:

```
RealPosition = AddGadget(Window, Gadget, Position);
     D0          -42        A0       A1       D0
```

We first pass the window in which the gadget should be inserted along with the address of the gadget itself. Next we test for the position that it should take in the list. As a result we get the current position.

For our toggle gadget the command looks like this:

```
Pos = AddGadget(FirstWindow, ToggleGadget, -1L);
```

The -1 states that the gadget should be inserted as the last thing in the list.

With this function call the gadget takes control in the window list, but it cannot yet be seen by the user. It still needs an additional function call, called automatically when a new window is set up. We call this function using its name:

```
RefreshGadgets(Gadgets, Window, Requester);
   -222          A0        A1        A2
```

This function shows all of the given gadgets in the window list. Not everything new is displayed with this; we test with gadgets to see which gadget the routine should begin with. That is why we placed our new gadget at the end of the list. We enter its number, and this is displayed.

We insert RefreshGadget() following the AddGadget() statement:

```
RefreshGadgets(Pos, FirstWindow, NULL);
```

This makes the gadget visible. Now let's look at the toggle gadget check:

```
/* 3.4.2.1.B.toggle_forever */
   FOREVER
      {
      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
         Wait(1L << FirstWindow->UserPort->mp_SigBit);
         continue;
         }
```

```
        MessageClass = message->Class;
        code = message->Code;
        GadgetPtr = (struct Gadget *) message->IAddress;
        SelectMode= GadgetPtr->Flags & SELECTED;

        ReplyMsg(message);
        switch (MessageClass)
           {
           case GADGETUP    : if (SelectMode)
                                 {
                                 printf("Activated ");
                                 }
                              else
                                 {
                                 printf("Deactivated\n");
                                 }
                              break;

           case CLOSEWINDOW : Close_All();
                              exit(TRUE);
                              break;
           }
        }
```

The test to see if a gadget was pressed is done with a case statement. The gadget status is also analyzed to see if it is "selected" or "unselected". A simple AND comparison does this. The program also displays the status of the gadget in the CLI window. Here is the complete program listing:

```
/*************************************
 *                                   *
 * Program:  Toggle-gadget.c         *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1988  for testing   *
 *                     only          *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


SHORT GadgetPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
```

```
    };

struct Border GadgetBorder =
    {
    -1, -1, 1, 0, JAM1, 5, GadgetPairs, NULL,
    };

struct IntuiText GadgetText =
    {
    2, 0, JAM2, 4, 7, NULL, (UBYTE *)"Gadget" ,NULL,
    };

struct Gadget BoolGadget =
    {
    NULL,                  /* NextGadget          */
    10, 40,                /* LeftEdge, TopEdge   */
    60, 20,                /* Width, Height       */
    GADGHCOMP,             /* Flags               */
    TOGGLESELECT,          /* Activation          */
    BOOLGADGET,            /* Gadget Type         */
    (APTR)&GadgetBorder,   /* GadgetRender        */
    NULL,                  /* Select Render       */
    &GadgetText,           /* GadgetText          */
    NULL,                  /* MutualExclude       */
    NULL,                  /* SpecialInfo         */
    1,                     /* GadgetID            */
    NULL,                  /* UserData            */
    };

struct NewWindow FirstNewWindow =
    {
    160, 50,               /* LeftEdge, TopEdge   */
    320, 150,              /* Width, Height       */
    0, 1,                  /* DetailPen, BlockPen */
    CLOSEWINDOW |          /* IDCMP Flags         */
    GADGETUP,

    WINDOWDEPTH |          /* Flags               */
    WINDOWSIZING |
    WINDOWDRAG |
    WINDOWCLOSE |
    SMART_REFRESH,

    &BoolGadget,           /* First Gadget        */
    NULL,                  /* CheckMark           */

    (UBYTE *)"System programmng test",

    NULL,                  /* Screen              */
    NULL,                  /* BitMap              */

    100, 50,               /* Min Width, Height   */
    640, 200,              /* Max Width, Height   */

    WBENCHSCREEN,          /* Type                */
    };
```

```
main()
   {
   ULONG MessageClass;
   USHORT code;

   struct Message *GetMsg();

   Open_All();

   RefreshGadgets(&BoolGadget, FirstWindow, NULL);


   FOREVER
      {
      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
         Wait(1L << FirstWindow->UserPort->mp_SigBit);
         continue;
         }
      MessageClass = message->Class;
      code = message->Code;
      GadgetPtr = (struct Gadget *) message->|Address;
      SelectMode = GadgetPtr->Flags & Selected;

      ReplyMsg(message);
      switch (MessageClass)
         {
         case GADGETUP  :  if (SelectMode)
                              {
                                 printf("Activated\n");
                              }
                           else
                              {
                                 printf("Deactivated\n");
                              }
                           break;

         case CLOSEWINDOW : Close_All();
                            exit(TRUE);
                            break;
         }
      }
   }


Open_All()
   {
   void *OpenLibrary();
   struct Window *OpenWindow();

   if (!(IntuitionBase = (struct IntuitionBase *)
         OpenLibrary("intuition.library", 0L)))
      {
```

```
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window
*)OpenWindow(&FirstNewWindow)))
      {
      printf("Window will not open!\n");
      Close_All();
      exit(FALSE);
      }
   }

Close_All()
   {
   if (FirstWindow)      CloseWindow(FirstWindow);
   if (IntuitionBase)    CloseLibrary(IntuitionBase);
   }
```

At the moment this new toggle gadget has no further function. What can turn this gadget on and off? Another gadget, of course. We'll add a new gadget and let it turn the toggle gadget on and off. This way we learn about two new functions.

As the second gadget we choose a completely normal boolean gadget to minimize programming hassles. Here's the structure:

```
/* 3.4.2.1.C secondgadget */
SHORT SelectPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
   };

struct Border SelectBorder =
   {
   -1, -1, 1, 0, JAM1, 5, SelectPairs, NULL,
   };

struct IntuiText SelectText =
   {
   1, 0, JAM2, 4, 7, NULL, (UBYTE *)"Select" ,NULL,
   };

struct Gadget SelectGadget =
   {
   NULL,                    /* NextGadget          */
   10, 40,                  /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
   GADGHCOMP |              /* Flags               */
   GADGDISABLED,
   RELVERIFY,               /* Activation          */
   BOOLGADGET,              /* Gadget Type         */
   (APTR)&SelectBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
```

```
    &SelectText,              /* GadgetText          */
    NULL,                     /* MutualExclude       */
    NULL,                     /* SpecialInfo         */
    2,                        /* GadgetID            */
    NULL,                     /* UserData            */
    };
```

(If you want you can use another gadget instead)

Structure description

As usual, we have supplied you with the border and the `IntuiText` structure. We now turn our attention to the `Select` gadget, which is the same as a normal boolean gadget with the exception that it is "disabled." It is set in this condition using the `GADGDISABLED` flag; the user cannot choose it. The entire click area is displayed in a graphic status called "ghosted."

This second gadget must also be added to the window. We can construct a second `AddGadget()` function. Intuition supplies another function in case more gadgets must be added. The entire list of gadgets can be added to the window with `AddGList()`. Set a pointer to the second gadget in the first gadget. Now, you can insert both gadgets into the window list using the `RealPosition()` function. The format for `RealPosition()`:

```
RealPosition = AddGList(Window, Gadget, Position, Numgad,
     D0         -438      A0       A1       D0        D1

     Requester);
        D2
```

The first three arguments may be familiar to you from the simple functions. By using `Numgad` you can state how many gadgets the list contains, and `Requester` must point to the requester when you need one. In this case, though, the value is stored with null.

The same problem that we had with `AddGadget()` is also encountered with `RefreshGadgets()`. We find the first gadget's position with `AddGList()`. We state this with the `Refresh` function. Only two gadgets should be newly displayed. For this reason there is also a list refresh command:

```
RefreshGList(Gadgets, Window, Requester, Numgad);
     -432      A0        A1        A2        D0
```

Similar to the `AddGList` function, `Numgad` specifies the number of gadgets a `Refresh` should execute.

Now we turn back to our program. It should return the boolean gadget as free because the toggle gadget was activated and returned. To do this we must be able to differentiate between the self defined gadgets. `GadgetID` helps us here. The number entered there makes it possible for us to identify each gadget. This is why you choose the different numbers for each gadget in the program.

Consequently the check must now be changed. You should familiarize yourself with the two functions that you can use to influence the status of a gadget. They are:

```
OffGadget(gadget, Window, Requester);
  -174      A0      A1       A2

OnGadget(gadget, Window, Requester);
 -186      A0      A1       A2
```

Both functions need the same arguments: a pointer to the gadget whose status should be changed, the pointer to the window in which the gadget is found and a pointer to a requester, if it is present. The last variable does not concern us at the moment.

Change the check so that when the `SELECT` flag is positive the program calls `OnGadget()`, and when the flag is negative, the program calls `OffGadget()`. Every time you pass a pointer to the `SELECT` gadget:

```
switch (GadgetNr)
   {
   case 2     : printf("selected!\n");
                break;

   case 1     : if (selectmode)
                    {
                    printf("activated ");
          OnGadget(&SelectGadget, FirstWindow,NULL);
          RefreshGadgets(&SelectGadget, FirstWindow,NULL);
                    }
                else
                    {
                     printf("deactivated\n");
          OffGadget(&SelectGadget, FirstWindow,NULL);
                     }
                break;
}
break;
```

Insert this instead of the `if` check by the `case GADGETUP`. In addition you must calculate the `ushort` variable `GadgetNr` before this point:

```
GadgetNr = GadgetPrt->GadgetID;
```

How it works When you have added all of the new sections to the first program, you should see the following when the program is run:

A window with two gadgets appear. The right gadget turns the left gadget on and off. The second gadget first appears as "ghosted." When you click on the right gadget, the type of the other gadget should be readable, not ghosted. Now you can select this gadget as well. The program shows its reactions using text in the CLI window. The text

"activated" and "deactivated" appear for the toggle gadget. When you can click on the left gadget, the program sends the text "selected!"

Here is the complete listing of the program:

```
/*************************************
 *                                   *
 * Program : OnOff-gaggets.c         *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1988  for testing   *
 *                     only          *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


SHORT SelectPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
   };

struct Border SelectBorder =
   {
   -1, -1, 1, 0, JAM1, 5, SelectPairs, NULL,
   };

struct IntuiText SelectText =
   {
   1, 0, JAM2, 4, 7, NULL, (UBYTE *)"Select" ,NULL,
   };

struct Gadget SelectGadget =
   {
   NULL,                 /* NextGadget          */
   10, 40,               /* LeftEdge, TopEdge   */
   60, 20,               /* Width, Height       */
   GADGHCOMP,            /* Flags               */
   RELVERIFY,            /* Activation          */
   BOOLGADGET,           /* Gadget Type         */
   (APTR)&SelectBorder,  /* GadgetRender        */
   NULL,                 /* Select Render       */
   &SelectText,          /* GadgetText          */
   NULL,                 /* MutualExclude       */
   NULL,                 /* SpecialInfo         */
```

```
   2,                    /* GadgetID             */
   NULL,                 /* UserData             */
   };

SHORT GadgetPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0
   };

struct Border GadgetBorder =
   {
   -1, -1, 1, 0, JAM1, 5, GadgetPairs, NULL
   };

struct IntuiText BoolText =
   {
   3, 0, JAM2, 4, 7, NULL, (UBYTE *)"BoolGad" ,NULL
   };

struct Gadget BoolGadget =
   {
   &SelectGadget,        /* NextGadget           */
   120, 40,              /* LeftEdge, TopEdge    */
   60, 20,               /* Width, Height        */
   GADGHBOX,             /* Flags                */
   RELVERIFY |           /* Activation           */
   TOGGLESELECT,
   BOOLGADGET,           /* Gadget Type          */
   (APTR)&GadgetBorder,  /* GadgetRender         */
   NULL,                 /* Select Render        */
   &BoolText,            /* GadgetText           */
   NULL,                 /* MutualExclude        */
   NULL,                 /* SpecialInfo          */
   1,                    /* GadgetID             */
   NULL                  /* UserData             */
   };

struct NewWindow FirstNewWindow =
   {
   160, 50,              /* LeftEdge, TopEdge    */
   320, 150,             /* Width, Height        */
   0, 1,                 /* DetailPen, BlockPen  */
   CLOSEWINDOW |         /* IDCMP Flags          */
   GADGETUP |
   GADGETDOWN,

   WINDOWDEPTH |         /* Flags                */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,

   &BoolGadget,          /* First Gadget         */
   NULL,                 /* CheckMark            */

   (UBYTE *)"System programming test",
```

```
    NULL,                   /* Screen                */
    NULL,                   /* BitMap                */

    100, 50,                /* Min Width, Height     */
    640, 200,               /* Max Width, Height     */

    WBENCHSCREEN            /* Type                  */
    };


main()
    {
    ULONG MessageClass;
    USHORT code, selectmode, GadgetNr;

    struct Message *GetMsg();
    struct Gadget *GadgetPtr;

    Open_All();

    RefreshGadgets(&BoolGadget, FirstWindow, NULL);


    FOREVER
        {
        if ((message = (struct IntuiMessage *)
             GetMsg(FirstWindow->UserPort)) == NULL)
            {
            Wait(1L << FirstWindow->UserPort->mp_SigBit);
            continue;
            }
        MessageClass = message->Class;
        code = message->Code;
        GadgetPtr = (struct Gadget *) message->IAddress;
        GadgetNr  = GadgetPtr->GadgetID;
        selectmode= GadgetPtr->Flags & SELECTED;

        ReplyMsg(message);
        switch (MessageClass)
            {
            case GADGETUP    :

            case GADGETDOWN  :

                switch (GadgetNr)
                    {
                    case 2      : printf("selected!\n");
                                  break;

                    case 1      : if (selectmode)
                                      {
                                      printf("activated ");
                                      OnGadget(&SelectGadget,
FirstWindow,NULL);
```

```
RefreshGadgets(&SelectGadget, FirstWindow,NULL);
                                    }
                                 else
                                    {
                                    printf("deactivated\n");
                                    OffGadget(&SelectGadget,
FirstWindow,NULL);
                                    }
                                 break;
                  }
                  break;

         case CLOSEWINDOW : Close_All();
                            exit(TRUE);
                            break;
         }
      }
   }


Open_All()
   {
   void *OpenLibrary();
   struct Window *OpenWindow();

   if (!(IntuitionBase = (struct IntuitionBase *)
         OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window
*)OpenWindow(&FirstNewWindow)))
      {
      printf("Window will not open!\n");
      Close_All();
      exit(FALSE);
      }
   }

Close_All()
   {
   if (FirstWindow)      CloseWindow(FirstWindow);
   if (IntuitionBase)    CloseLibrary(IntuitionBase);
   }
```

Now that we've finished with boolean gadgets, let's describe the other two types. First we should turn our attention to the proportional gadgets, for which there are two special functions. This example will supply a graphic to the knob. Here is the pertinent `Image` structure with the graphic data:

```
USHORT KnobGraphic[] =
   {
   0xCCCC,
   0x6666,
   0x3333,
   0x9999,
   0xCCCC,
   0x6666,
   0x3333,
   0x9999,
   0xCCCC,
   0x6666,
   0x3333,
   0x9999,
   0xCCCC,
   0x6666,
   0x3333,
   0x9999
   };

struct Image Knob =
   {
   0, 0,
   16, 16,
   1,
   &KnobGraphic[0],
   1, 0,
   NULL
   };
```

The gadget structure with the `PropInfo` belongs to this gadget:

```
struct PropInfo SliderProp =
{
   FREEVERT,            /*    Flags           */
   0x8000,              /*    HorizPot        */
   0x8000,              /*    VertPot         */
   0x0800,              /*    HorizBody       */
   0x1000,              /*    VertBody        */
   0,                   /*    CWidth          */
   0,                   /*    CHeight         */
   0,                   /*    HPotRes         */
   0,                   /*    VPotRes         */
   0,                   /*    LeftBorder      */
   0                    /*    TopBorder       */
};

struct Gadget Slider =
   {
   NULL,                /* NextGadget         */
   260, 40,             /* LeftEdge, TopEdge  */
   24, 120,             /* Width, Height      */
   GADGHNONE |          /* Flags              */
   GADGIMAGE,
   RELVERIFY,           /* Activation         */
   PROPGADGET,          /* Gadget Type        */
```

```
   (APTR)&Knob,          /* GadgetRender        */
   NULL,                 /* Select Render       */
   NULL,                 /* GadgetText          */
   NULL,                 /* MutualExclude       */
   (APTR)&SliderProp,    /* SpecialInfo         */
   1,                    /* GadgetID            */
   NULL,                 /* UserData            */
   };
```

Note: The graphic data of the mover must be stored in chip memory or it will not be displayed, use the +C option of the Aztec compiler to place this data in chip memory.

We should read the mover's position. Insert these two lines in the gadget check:

```
case GADGETUP :printf("Mover position: %x/n",
moveprop.VertPot);
  break;
```

You can check and evaluate the slider's position in the program using these methods. Here is a complete example program:

```
/*************************************
 *                                   *
 * Program: Prop-Image-Gadget.c      *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1988  for testing   *
 *                     only, place in *
 *                     chip ram      *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Screen        *FirstScreen;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


USHORT KnobGraphic[] =
   {
   0xCCCC,
   0x6666,
   0x3333,
   0x9999,
   0xCCCC,
   0x6666,
   0x3333,
```

```
    0x9999,
    0xCCCC,
    0x6666,
    0x3333,
    0x9999,
    0xCCCC,
    0x6666,
    0x3333,
    0x9999
    };

struct Image Knob =
    {
    0, 0,
    16, 16,
    1,
    &KnobGraphic[0],
    1, 0,
    NULL
    };


struct PropInfo SliderProp =
{
    FREEVERT,           /*    Flags          */
    0x8000,             /*    HorizPot       */
    0x8000,             /*    VertPot        */
    0x0800,             /*    HorizBody      */
    0x1000,             /*    VertBody       */
    0,                  /*    CWidth         */
    0,                  /*    CHeight        */
    0,                  /*    HPotRes        */
    0,                  /*    VPotRes        */
    0,                  /*    LeftBorder     */
    0                   /*    TopBorder      */
};

struct Gadget Slider =
    {
    NULL,               /* NextGadget        */
    260, 40,            /* LeftEdge, TopEdge */
    24, 120,            /* Width, Height     */
    GADGHNONE |         /* Flags             */
    GADGIMAGE,
    RELVERIFY,          /* Activation        */
    PROPGADGET,         /* Gadget Type       */
    (APTR)&Knob,        /* GadgetRender      */
    NULL,               /* Select Render     */
    NULL,               /* GadgetText        */
    NULL,               /* MutualExclude     */
    (APTR)&SliderProp,  /* SpecialInfo       */
    1,                  /* GadgetID          */
    NULL,               /* UserData          */
    };
```

```
struct NewWindow FirstNewWindow =
   {
   160, 0,              /* LeftEdge, TopEdge    */
   320, 200,            /* Width, Height        */
   0, 1,                /* DetailPen, BlockPen */
   CLOSEWINDOW |         /* IDCMP Flags          */
   GADGETUP,
   WINDOWDEPTH |        /* Flags                */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   &Slider,             /* First Gadget         */
   NULL,                /* CheckMark            */
   (UBYTE *)"Proportional-Gadget",
   NULL,                /* Screen               */
   NULL,                /* BitMap               */
   100, 50,             /* Min Width, Height    */
   640, 200,            /* Max Width, Height    */
   WBENCHSCREEN,        /* Type                 */
   };


main()
   {
   ULONG MessageClass;
   USHORT code;

   struct Message *GetMsg();

   Open_All();

   FOREVER
      {
      if (message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort))
         {
         MessageClass = message->Class;
         code = message->Code;
         ReplyMsg(message);
         switch (MessageClass)
            {
            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;
            case GADGETUP :printf("Mover position: %x/n",
moveprop.VertPot);
                               break;

            }
         }
      }
   }
```

```
Open_All()
   {
   void          *OpenLibrary();
   struct Window *OpenWindow();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&FirstNewWindow)))
      {
      printf("Window will not open!\n");
      Close_All();
      exit(FALSE);
      }
   }


Close_All()
   {
   if (FirstWindow)     CloseWindow(FirstWindow);
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

There are situations in which it is important to change the position of the slider during the program (e.g., size or something similar). We would like to describe one method which can be used with any gadget. Then we will explain two functions that are used only for proportional gadgets.

General tip for all gadgets: To change any setting on a gadget, you must first remove this gadget from the window list. When this is not done, errors occur. `RemoveGadget()` is a function which permits a gadget to be removed from the list.

```
Position = RemoveGadget(Window, Gadget);
  eD0         -228          A0       A1
```

Now you can make changes to the `Gadget` structure, the `PropInfo` structure and all associated structures. Then it can be added back to the list with `AddGadget()`. To conclude you must update the screen using `RefreshGadget()`.

In other cases where entire portions of gadgets are removed, changed and then reentered in the list, there is an additional function:

```
Position = RemoveGList(Window, Gadget, Numgad);
   De0       -444         A0       A1       D0
```

Here you give the number of gadgets that should be removed. The opposite, `AddGList()`, has already been discussed. Don't forget to update the graphic with `RefreshGList()`.

The second way, only for proportional gadgets, saves the time of deleting and reinserting every gadget from the list. The `ModifyProp()` function lets you change all possible gadget settings in the `PropInfo` structure, after describing the gadget. The gadget is then automatically redisplayed:

```
ModifyProp(Gadget, Window, Requester, Flags, HorizPot,
  -156       A0       A1        A2        D0       D1

     VertPot, HorizBody, VertBody);
       D2         D3         D4
```

Unfortunately this function has a disadvantage: In most applications it is often necesary to change the settings using `ModifyProp()`. However, when `RefreshGadgets()` performs it function, all the gadgets following in the list are also "refreshed." This can be annoying, therefore `NewModifyProp()` was added to the Intuition functions.

```
NewModifyProp(Gadget, Window, Requester, Flags, HorizPot,
  -156          A0       A1        A2        D0       D1

     VertPot, HorizBody, VertBody, Numgad);
       D2         D3         D4        D5
```

The new function serves exactly the same purpose as `ModifyProp`. You can use `NewModifyProp()` to select how many gadgets in the list should be refreshed. We recommend 1 as a good value, unless you have other gadgets which relate to the change and should be changed accordingly.

File Selector

We would now like to present the structure for a file selector box, used to select a single file name from a list of many file names. This is used to load a file into an application program. Examples of these are loading pictures into a drawing program or loading a text file into an editor.

If you thought that you could print out the names with the `Intuitext` structure, in conjunction with the `Gadget` structure, you're wrong. The programming can get rather complicated when a file list is present that contains more names that can fit in the window. The text must be scrolled to allow the user to see all of the files.

It would be best to create a routine to display the list of names, beginning with a certain one, in a predetermined order. The number of names that are to be displayed is the number of gadgets that have no graphics associated with them. They are only known by their click area and the

text displayed in them. This cannot be more than one line, 8/9 points, and must be within the maximum number of characters of a filename. Because these are all boolean gadgets, we can use the simple gadget structure:

```
/*3.4.2.1.D.filegadget_struct*/
struct Gadget FileGadget =
   {
   NULL,                    /* NextGadget          */
   80, 0,                   /* LeftEdge, TopEdge   */
   200, 8,                  /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   BOOLGADGET |
   REQGADGET,               /* Gadget Type         */
   NULL,                    /* GadgetRender        */
   NULL,                    /* Select Render       */
   NULL,                    /* GadgetText          */
   NULL,                    /* MutualExclude       */
   NULL,                    /* SpecialInfo         */
   0,                       /* GadgetID            */
   NULL,                    /* UserData            */
   };
```

Because of the large width of your gadget we would like to ask that you now load the `First_window.c` program from Section 3.1 again and change the window structure there so that the window has a `LeftEdge` of 120 and `Width` of 360. We want to begin developing a file select box, and we need a wide enough window to do so. Look at the design:

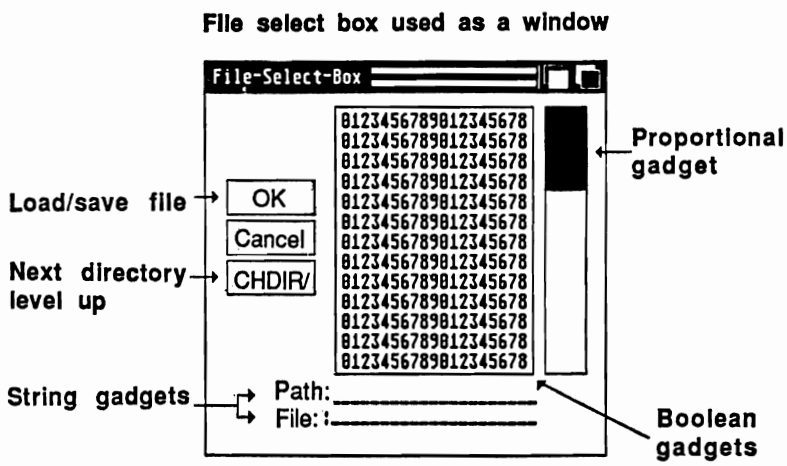


Figure 3.11

You also need a large field for the filenames, a scroll bar (which you programmed above), two string gadgets and three boolean gadgets. After these digressions for an overview of your project, we turn back to the many gadgets for selecting a name.

With the gadget structure defined first we don't get very far. You need something more, defining every structure individually is too hard. Remember the example of the `IntuiText` structure. There we reproduced a structure quite simply. That would be handy to have here:

Preceding the `main` function we define this gadget structure and a field with unfilled structures.

```
struct Gadget Files[15];
```

We need a function in the program that copies the general structure and at the same time corrects the Y position.

```
/*3.4.2.1.E.makefiles*/
Make_Files(Structur)
struct Gadget Struktur[];

   {
   int i;

   for (i=0; i<15; i++)
      {
      Struktur[i]             = FileGadget;
      Struktur[i].TopEdge     = 30 + i * 8;
      Struktur[i].GadgetID    = i + 1;
      Struktur[i].NextGadget  = &Struktur[i+1];
      }
   Struktur[14].NextGadget = NULL;
   }
```

How it works The function puts the general structure in the structure field and corrects the Y value (`TopEdge`). In addition it links the entire list of 15 gadgets together so that you can move them into the window with a simple call (`AddGList()`). Furthermore, every gadget receives an ID beginning with 1. This lets you establish later which of the user gadgets were clicked on. A `Refresh` is unnecessary in this case because no graphic is used.

Now insert the above defined proportional gadget into the program. Change the variable `LeftEdge` to a value of 300. The ID of these gadgets must also receive a value greater than 15 so that it is differentiated from the others. We recommend 20, so there is space for enlarging.

You are now missing the two strings. Because there are only two, you can define them normally through the structure. Here we encounter the wonderful condition that we only need one `Undo` buffer because only one gadget can be active at a time.

```
/* 3.4.2.1.F.buffers*/
unsigned char PathBuffer[512] = "df1:";
unsigned char FileBuffer[31]   = "";
unsigned char UndoBuffer  [512];

struct StringInfo PathInfo =
   {
   &PathBuffer[0],          /* Buffer              */
   &UndoBuffer[0],          /* Undo Buffer         */
   5,                       /* Buffer Position     */
   511,                     /* MaxChars            */
   0,                       /* Display Position    */
   0,                       /* Undo Position       */
   0,                       /* NumChars            */
   0,                       /* Display Counter     */
   0, 0,                    /* CLeft, CTop         */
   NULL,                    /* LayerPtr            */
   0,                       /* LongInt             */
   NULL,                    /* AltKeyMap           */
   };

struct StringInfo FileInfo =
   {
   &FileBuffer[0],          /* Buffer              */
   &UndoBuffer[0],          /* Undo Buffer         */
   0,                       /* Buffer Position     */
   31,                      /* MaxChars            */
   0,                       /* Display Position    */
   0,                       /* Undo Position       */
   0,                       /* NumChars            */
   0,                       /* Display Counter     */
   0, 0,                    /* CLeft, CTop         */
   NULL,                    /* LayerPtr            */
   0,                       /* LongInt             */
   NULL,                    /* AltKeyMap           */
   };

SHORT StringPairs[] =
   {
   0, 0, 248, 0
   };

struct Border StringBorder =
   {
   0, 9, 1, 0, JAM1, 2, StringPairs, NULL,
   };

struct IntuiText PathText =
   {
   1, 0, JAM2, -58, 1, NULL, (UBYTE *)"Path:" ,NULL,
   };

struct IntuiText FileText =
   {
   1, 0, JAM2, -58, 1, NULL, (UBYTE *)"File :" ,NULL,
   };
```

```
struct Gadget Path =
   {
   NULL,                    /* NextGadget          */
   80, 150,                 /* LeftEdge, TopEdge   */
   250, 8,                 /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   STRGADGET,               /* Gadget Type         */
   (APTR)&StringBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &PAthText,              /* GadgetText          */
   NULL,                    /* MutualExclude       */
   (APTR)&PathInfo,        /* SpecialInfo         */
   21,                      /* GadgetID            */
   NULL,                    /* UserData            */
   };

struct Gadget File =
   {
   &Path,                  /* NextGadget          */
   80, 165,                 /* LeftEdge, TopEdge   */
   250, 8,                 /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   STRGADGET,               /* Gadget Type         */
   (APTR)&StringBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &FileText,              /* GadgetText          */
   NULL,                    /* MutualExclude       */
   (APTR)&FileInfo,        /* SpecialInfo         */
   22,                      /* GadgetID            */
   NULL,                    /* UserData            */
   };
```

You need boolean gadgets as the last gadgets for your file select box. You simply need one border structure, three `IntuiText` structures, and three gadget structures:

```
/*3.4.2.1.G.finalgadgets*/
SHORT SelectPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
   };

struct Border SelectBorder =
   {
   -1, -1, 1, 0, JAM1, 2, StringPairs, NULL,
   };

struct IntuiText OKText =
   {
   1, 0, JAM2, 22, 5, NULL, (UBYTE *)"  OK   " ,NULL,
   };
```

```
struct IntuiText CancelText =
   {
   1, 0, JAM2, 2, 5, NULL, (UBYTE *)"Cancel" ,NULL,
   };

struct IntuiText ReturnText =
   {
   1, 0, JAM2, 2, 5, NULL, (UBYTE *)"CHDIR" ,NULL,
   };

struct Gadget OK =
   {
   NULL,                    /* NextGadget          */
   10, 40,                  /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   BOOLGADGET,              /* Gadget Type         */
   (APTR)&SelectBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &OKText,                 /* GadgetText          */
   NULL,                    /* MutualExclude       */
   NULL,                    /* SpecialInfo         */
   23,                      /* GadgetID            */
   NULL,                    /* UserData            */
   };

struct Gadget Cancel =
   {
   &OK,                     /* NextGadget          */
   10, 65,                  /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   BOOLGADGET,              /* Gadget Type         */
   (APTR)&SelectBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &CancelText,             /* GadgetText          */
   NULL,                    /* MutualExclude       */
   NULL,                    /* SpecialInfo         */
   24,                      /* GadgetID            */
   NULL,                    /* UserData            */
   };

struct Gadget Return =
   {
   &Cancel,                 /* NextGadget          */
   10, 90,                  /* LeftEdge, TopEdge   */
   60, 20,                  /* Width, Height       */
   GADGHCOMP,               /* Flags               */
   RELVERIFY,               /* Activation          */
   BOOLGADGET,              /* Gadget Type         */
   (APTR)&SelectBorder,     /* GadgetRender        */
   NULL,                    /* Select Render       */
   &ReturnText,             /* GadgetText          */
   NULL,                    /* MutualExclude       */
```

```
   NULL,                        /* SpecialInfo          */
   25,                          /* GadgetID             */
   NULL,                        /* UserData             */
   };
```

Now you have three linked gadget lists: the file gadgets without graphics, the two strings gadgets, and the boolean gadgets. You also have a proportional gadget. Now construct all of the gadget definitions in the standard program and think about the window measurements. Then add the following after the `Open_All()` function call in the `main` function:

```
/* 3.4.2.1.H.makingmorefiles */
Make_Files(Files);

FilePos = AddGList(FirstWindow, Files, -1L, 15, NULL);
StrgPos = AddGList(FirstWindow, File, -1L, 2, NULL);
SlctPos = AddGList(FirstWindow, Return, -1L, 3, NULL);
PropPos = AddGadget(FirstWindow, Slider, -1L);

RefreshGList(File, FirstWindow, NULL, 2);
RefreshGList(Return, FirstWindow, NULL, 3);
RefreshGList(Slider, FirstWindow, NULL, 1);
```

We don't want to give the gadget check an evaluation. Here we insert an output of the `GadgetID` in case it is handled as being self defined. The important program sections are found in the example numbered 3.4.2.1.c.1. It is missing the line:

```
printf("Gadget number: %d\n", GadgetNr);
```

A complete program of this file selector is presented in Section 3.5.4.

3.4.2.2 A new sizing gadget

Now we finally come to a custom sizing gadget. When you click on this gadget, and the window is not at the maximum size that it can assume on the screen, it assumes full size. When it reaches the maximum size on the screen, clicking it again brings the size to its minimum. Be careful that the `MinWidth` and `MinHeight` values are initialized.

```
/*3.4.2.2.A.sizing*/
struct Gadget Sizing =
   {
   NULL,                        /* NextGadget           */
   -10, 15,                     /* LeftEdge, TopEdge    */
   8, 10,                       /* Width, Height        */
   GADGHCOMP |                  /* Flags                */
   GRELRIGHT,
```

```
   RELVERIFY |                  /* Activation          */
   RIGHTBORDER,
   BOOLGADGET,                  /* Gadget Type         */
   (APTR)&SizingImage,          /* GadgetRender        */
   NULL,                        /* Select Render       */
   NULL,                        /* GadgetText          */
   NULL,                        /* MutualExclude       */
   NULL,                        /* SpecialInfo         */
   0,                           /* GadgetID            */
   NULL,                        /* UserData            */
   };
```

As the Image structure, we ask you to use the structure that has been implemented at the end of the preceding section. The check routine should have the following modifications:

```
/*3.4.2.2.b.modifications */
Sizing()
{
SHORT ScreenWidth, ScreenHeight;

ScreenWidth = Window->WScreen->Width;
ScreenHeitgh= Window->WScreen->Height;

if (Window->LeftEdge == NULL & Window->TopEdge == NULL)
   if (Window->Width == ScreenWidth &
                          Window->Height == ScreenHeight)
    SizeWindow(Window, -1*(Window->Width -
                    Window->MinWidth),

                         -1*(Window->Height -
                              Window->MinHeight);

   MoveWindow(Window, -1*Window->LeftEdge,
                   -1*Window->TopEdge);
   SizeWindow(Window, ScreenWidth - Window->Width,
                   ScreenHeight - Window->Height);
}
```

3.5 Requesters

Requesters are an extension of gadgets. Gadgets are used by programs to request information and user selections during program execution.

Sometimes a program must request information or user input immediately during its execution. The computer must then ask the user for the information needed, before the program can execute further.

A requester appears to request this information. A requester is a box containing at least one gadget to aid user response. Text can be added to the gadgets to clarify the problem or question. If the programmer prefers, he can construct a requester entirely of graphics.

System requester

The simplest example is the system requester. They appear when a certain disk is required by the operating system. This requester allows you to either Retry or Cancel the operation.

We want to place the file select box window we created in the previous section in a requester. This requester will ask whether the user wants to load or save a file. The user needs information such as the filename and the directory in which it can/should be found.

We mentioned that requesters were supported by Intuition. You can also draw your own fairly easily.

3.5.1 Automatic requesters

Requesters are a collection of many gadgets. You have already seen that gadget programming is simple but requires many parameters.

Intuition offers a simple solution to this problem: Use the automatic requester. Automatic requesters consist of two gadgets and one prompting text. This simple requester doesn't require any complex programming. The greatest advantage to the automatic requester is it doesn't need a structure of its own. You simply give the function the pointer to the `IntuiText` structure and you get a value back when the user responds to the requester.

You need a pointer to a window to work with your automatic requester. This must not be initialized.

Next three `IntuiText` structures can be given. The first represents a general text used for an explanation of the problem (i.e., an explaination or question). The other two texts act as gadget texts, as you saw from the system requesters. The left corner states the positive text, listing a consent or confirmation such as Retry. The right corner states the negative text, listing denial, rejection or cancellation, such as Cancel.

We are still missing two flags. The first represents the positive gadget. You can set this flag at zero because clicking on the gadget activates the positive condition. The second flag does the same job as the first, except it represents the negative gadget. These last two values are known. Because Intuition doesn't know the measurements of the text and the gadgets, you can set the requester's size at this point. Intuition calculates the border sizes for the gadget text—so you don't have to.

Here's the general format of our new function with all of the arguments in one short description:

```
Response = AutoRequest(Window, BodyText, PositiveText,
   D0          -348      A0       A1          A2
                NegativeText, PositiveFlags, NegativeFlags,
                     A3           D0             D1
                Width, Height);
                  D2     D3
```

Arguments

`Window`	Pointer to a `Window` structure.
`BodyText`	`IntuiText` general text structure.
`PositiveText`	`IntuiText` for Yes gadget.
`NegativeText`	`IntuiText` for No gadget.
`PositiveFlags`	`IDCMP` flag for Yes gadget.
`NegativeFlags`	`IDCMP` Flag for No gadget.
`Width`	Width of auto requester window.
`Height`	Height of auto requester window.

The requester example does not need the `First_window.c` program. You simply initialize the `IntuiText` structures and assign all of the values of the function. You get the response of the user by checking the positive gadget. The following program reads it and displays a corresponding text in the CLI window.

```
/*********************************
 *                               *
 * Program: AutoRequester.c      *
 * ============================= *
 *                               *
 * Author:  Date:      Comments: *
 * ------  ----------  ---------- *
 * Wgb     12/12/1987            *
 *                               *
 *                               *
 *********************************/


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


struct IntuitionBase *IntuitionBase;
struct Window        *Window = NULL;


struct IntuiText Body =
   {
   0, 1,
   JAM2,
   10, 10,
   NULL,
   (UBYTE *)"Do you wish to program requesters?",
   NULL
   };

struct IntuiText YES =
   {
   2, 3,
   JAM2,
   5, 3,
   NULL,
   (UBYTE *)" I do !!!",
   NULL
   };

struct IntuiText NO =
   {
   2, 3,
   JAM2,
   5, 3,
   NULL,
   (UBYTE *)" I don't !!!",
   NULL
   };


main()
   {
   BOOL Answer;
```

```
   Open_All();

   Answer = AutoRequest(Window, &Body, &YES, &NO, 0L, 0L,
320L, 60L);

   if (Answer == TRUE)
      printf("That is the truth!\n");
   else
      printf("The truth is always the best.\n");

   Close_All();
   }



Open_All()
   {
   void          *OpenLibrary();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }
   }



Close_All()
   {
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

How it works The special feature of this program is that we can display a system requester even if we don't have a reference window. This is important, for example, for the request to release memory when it is impossible to open the output window. To do this we set the window pointer to null. This is not possible with every requester function.

`AutoRequest` responds to clicking a gadget, pressing a key combination or inserting a disk. Pressing right <Amiga><V> selects the Retry gadget, while pressing right <Amiga><B> selects the Cancel gadget.

The auto requester window has one small disadvantage We can't define any features of the requester window, and the `AutoRequest` assigns the text, gadget graphics and dimensions.

3.5.2 System requesters

The use and programming of Auto-requesters is very simple. One drawback: There is no possibility for outside manipulation from the programmer.

The `AutoRequest()` function calls `BuildSysRequest()` which does allow modifications. The following stipulations must be met to work with `BuildSysRequest()`:

First, a window must be opened that should state the text size. Then we assign texts for both gadgets. In addition, we need the `IDCMP` flag for the requester window and the size of the requester.

The routine constructs a requester based on these arguments, in the same style as `AutoRequest`: Body text in the top section of the requester, gadgets in the lower left and right corners stating positive and negative responses. A communication channel returns the pointer to the requester window, either based on the pointer we assigned, or through parameters given by the Workbench.

We must write the checking routine for the new requester. We can rely on all IDCMP options for this routine. All input is permissible—it doesn't matter whether the input comes from the mouse or the keyboard.

When the routine has problems opening the requester, this requester changes into a recoverable Alert, just like an `AutoRequest`. Then `BuildSysRequest()` returns a true value to a window instead of a pointer. This tells us which of the two answers were selected.

Here's the format of this requester making function:

```
ReqWindow = BuildSysRequest(Window, BodyText,
    D0          -360          A0       A1
 PositiveText,  NegativeText, IDCMPFlags, Width, Height);
    A2              A3          D0        D2      D3

struct Window       *ReqWindow;
struct Window       *Window;
struct IntuiText    *BodyText;
struct IntuiText    *PositiveText;
struct IntuiText    *NegativeText;
ULONG  IDCMPFlags;
SHORT  Width, Height;
```

For your check routine you'll need some information about the gadgets. The following gadget flags should be set for each gadget:

```
BOOLGADGET, RELVERIFY, REQGADGET and TOGGLESELECT
```

The gadgets should look the same, just to keep things consistent. Use the AUTO flags in the <intuition/intuition.h> include file:

Auto-Name	Value	Description
AUTODRAWMODE	JAM2	Background and foreground color
AUTOFRONTPEN	0L	Character color 1
AUTOBACKPEN	1L	Character color 2
AUTOLEFTEDGE	6L	Positions of the IntuiTexts
AUTOTOPEDGE	3L	
AUTOITEXTFONT	NULL	Character sets for texts: Workbench-Zs
AUTONEXTTEXT	NULL	Pointer to further texts

All flags can be used in the IntuiText structures so that a uniform user interface is possible. You should inform the user if he must keep watch for other conventions.

With this description it should be possible for you to open a system requester. The only thing missing is the reaction when closing the requester. This is needed to free up all structures and memory that were used. This task is taken care of by the FreeSysRequest() function. It uses the pointer returned by the BuildSysRequest() routine. If only a true value is given, this function may not be called.

3.5.3 Making our own requester

Sometimes the auto requester just isn't enough. In many cases you need more than a yes/no answer. Enter the custom requester.

The system needs so many statements for a custom requester that it pays to set up a structure. This structure contains all of the information about the type, extent and appearance of the requester. By creating the structure we add the advantage of flexibility. We must program our own checking routines because the system no longer has access to its defaults for requesters, as it does for AutoRequest().

Think of a custom requester as a window into which we place multiple gadgets. At least one gadget tells the program that input has ended. Then we can analyze the data. This opens up custom requester design.

3.5.3.1 Requester structure

We need a requester structure for each requester. They contain information similar to that found in the Window structure:

```
struct Requester
{
0x00  00  struct Requester *OlderRequest;
0x04  04  SHORT LeftEdge;
0x06  06  SHORT TopEdge;
0x08  08  SHORT Width;
0x0A  10  SHORT Height;
0x0C  12  SHORT RelLeft;
0x0E  14  SHORT RelTop;
0x10  16  struct Gadget *ReqGadget;
0x14  20  struct Border *ReqBorder;
0x18  24  struct IntuiText *ReqText;
0x1C  28  USHORT Flags;
0x1E  30  UBYTE BackFill;
0x1F  31  struct Layer *ReqLayer;
0x23  35  UBYTE ReqPad1[32];
0x43  67  struct BitMap *ImageBMap;
0x47  71  struct Window *RWindow;
0x4B  75  UBYTE ReqPad2[36];
0x6F  111
};
```

The variable OlderRequester needs only to be initialized, then it can only be used by Intuition for management purposes.

LeftEdge, TopEdge, Width and Height state the position and size of the requester in the window.

RelLeft and RelTop serve a very useful function. Set the flag POINTREL in the flags value. Then the requester position is given relative to the mouse pointer position instead of at the LeftEdge and TopEdge positions. This way you can place a requester so that a certain gadget lies beneath the mouse pointer.

The next three pointers in the structure let you use Gadgets, Border and Texts. A requester cannot exist without gadgets. Border lets you can give the requester an attractive border, and IntuiText specifies the explanatory text.

The most important value of the structure, flags, is a flag. You can set the flag POINTREL from here so that the requester appears relative to the mouse pointer position. A second flag named PREDRAWN indicates that all Border, IntuiText and Image structures should be ignored. The ImageBMap takes over the entire graphic. Intuition searches for an initialized bit-map and fills it with graphics. Notice that

this new graphic also corresponds to the click area of the gadgets, because these no longer have their own graphic.

Intuition also sets its own flags in this value. The REQOFFWINDOW flag indicates that the requester is found outside of the bordering window, and REQACTIVE indicates when the requester is active. The flag SYSREQUEST is set when it is handled as a system requester (see AutoRequest and SystemRequest).

The next variable in the structure is BackFill. Here you find the number of the drawing pen that should fill in the background of the requester.

ReqLayer is filled from Intuition and contains a pointer to the layer that contains the requester.

ImageBMap is a pointer to a custom bit-map.

RWindow is a variable managed from Intuition. It points to the window where the requester appears.

3.5.3.2 Establishing a requester structure

Because of the difficulty in collecting the gadgets for a requester, you can add the example below to your own programs. First we place the file selector box in a requester. This makes the program handling simpler and more structured. You could use a file select box for our "big" project (the editor).

```
/* 3.5.3.2.a. setreqstructure */
struct Requester FileSelectBox;

InitRequest(&FileSelectBox);

FileSelectBox.LeftEdge = 2;
FileSelectBox.TopEdge  = 10;
FileSelectBox.Width    = 360;
FileSelectBox.Height   = 200;
FileSelectBox.ReqGadget= &Return;
```

Structure description

The measurements of the requester fit those of the window previously used for the file select box. The InitRequest() function set all arguments to null. Next we expand values. The gadget value points to the first gadget and moves to the NextGadgetPointer of each structure. That's why you must link the four gadget blocks. Because the REQGADGET flag must be set in each gadget of a requester, the example below takes you through all four gadget blocks:

```
/*3.5.3.2.b. reqgadgetset */
unsigned char PathBuffer[512] = "df1:";
unsigned char FileBuffer[31]   = "";
unsigned char UndoBuffer  [512];

USHORT KnobGraphic[] =
   {
   0xCCCC,   0x6666,   0x3333,   0x9999,
   0xCCCC,   0x6666,   0x3333,   0x9999,
   0xCCCC,   0x6666,   0x3333,   0x9999,
   0xCCCC,   0x6666,   0x3333,   0x9999
   };

struct Image Knob =
   {
   0, 0, 16, 16, 1, &KnobGraphic[0], 1, 0, NULL
   };

struct PropInfo SliderProp =
{
   FREEVERT,
   0x8000,   0x8000,
   0x0800,   0x1000,
   0,        0,
   0,        0,
   0,        0
};

struct StringInfo PathInfo =
   {
   &PathBuffer[0], &UndoBuffer[0],
   5, 511, 0, 0, 0, 0, 0, 0, NULL, 0, NULL
   };

struct StringInfo FileInfo =
   {
   &FileBuffer[0], &UndoBuffer[0],
   0, 31, 0, 0, 0, 0, 0, 0, NULL, 0, NULL
   };

SHORT StringPairs[] =
   {
   0, 0, 248, 0
   };

SHORT SelectPairs[] =
   {
   0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
   };

struct Border StringBorder =
   {
   0, 9, 1, 0, JAM1, 2, StringPairs, NULL,
   };

struct Border SelectBorder =
```

```
    {
    -1, -1, 1, 0, JAM1, 5, SelectPairs, NULL,
    };

struct IntuiText PathText =
    {
    1, 0, JAM2, -58, 1, NULL, (UBYTE *)"Path:" ,NULL,
    };

struct IntuiText FileText =
    {
    1, 0, JAM2, -58, 1, NULL, (UBYTE *)"File:" ,NULL,
    };

struct IntuiText OKText =
    {
    1, 0, JAM2, 22, 5, NULL, (UBYTE *)"OK" ,NULL,
    };

struct IntuiText CancelText =
    {
    1, 0, JAM2, 2, 5, NULL, (UBYTE *)"Cancel" ,NULL,
    };

struct IntuiText ReturnText =
    {
    1, 0, JAM2, 2, 5, NULL, (UBYTE *)"Return!" ,NULL,
    };

struct Gadget Files[15];
struct Gadget FileGadget =
    {
    NULL,
    80, 0, 240, 8,
    GADGHCOMP, RELVERIFY, BOOLGADGET | REQGADGET,
    NULL, NULL, NULL,
    NULL, NULL, 0, NULL
    };

struct Gadget Slider =
    {
    &Files[0],
    322, 30, 24, 120,
    GADGHNONE | GADGIMAGE, RELVERIFY, PROPGADGET |
REQGADGET,
    (APTR)&Knob, NULL, NULL,
    NULL, (APTR)&SliderProp, 1, NULL
    };

struct Gadget Path =
    {
    &Slider,
    80, 150, 250, 8,
    GADGHCOMP, RELVERIFY, STRGADGET | REQGADGET,
    (APTR)&StringBorder, NULL, &PathText,
    NULL, (APTR)&PathInfo, 21, NULL
```

```
    }; /*165 pal*/

struct Gadget File =
    {
    &Path,
    80, 165, 250, 8,
    GADGHCOMP, RELVERIFY, STRGADGET | REQGADGET,
    (APTR)&StringBorder, NULL, &FileText,
    NULL, (APTR)&FileInfo, 22, NULL
    };  /*185pal*/

struct Gadget OK =
    {
    &File,
    10, 40, 60, 20,
    GADGHCOMP, RELVERIFY | ENDGADGET , BOOLGADGET |
REQGADGET,
    (APTR)&SelectBorder, NULL, &OKText,
    NULL, NULL, 23, NULL
    };

struct Gadget Cancel =
    {
    &OK,
    10, 65, 60, 20,
    GADGHCOMP, RELVERIFY | ENDGADGET , BOOLGADGET |
REQGADGET,
    (APTR)&SelectBorder, NULL, &CancelText,
    NULL, NULL, 24, NULL
    };

struct Gadget Return =
    {
    &Cancel,
    10, 90, 60, 20,
    GADGHCOMP, RELVERIFY, BOOLGADGET | REQGADGET,
    (APTR)&SelectBorder, NULL, &ReturnText,
    NULL, NULL, 25, NULL
    };
```

3.5.3.3 The requester functions

We would now like to show you how to develop a program that uses the gadgets described above. This program makes it possible to use the file select box in real-world applications.

We need our `First_Window` program from Section 3.1. Define the `NewWindow` structure as printed, as well as the requester structure. Before this, we had to insert the gadgets with your "appendix." Now the program waits for a signal to activate a requester. We activate it. You

can insert a condition for it later, like a menu item selection, for example.

```
/* 3.5.3.3.a.requester additions */
struct NewWindow FirstNewWindow =
   {
   160, 0,              /* LeftEdge, TopEdge    */
   370, 200,             /* Width, Height        */
   0, 1,                 /* DetailPen, BlockPen */
   CLOSEWINDOW |         /* IDCMP Flags          */
   GADGETUP |
   REQCLEAR,
   WINDOWDEPTH |         /* Flags                */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                 /* First Gadget         */
   NULL,                 /* CheckMark            */
   (UBYTE *)"Requester Test",
   NULL,                 /* Screen               */
   NULL,                 /* BitMap               */
   100, 50,              /* Min Width, Height    */
   640, 200,             /* Max Width, Height    */
   WBENCHSCREEN,         /* Type                 */
   };
```

We use the simple Intuition function `Request()` to call the requester. It only needs two arguments. Here is the general format for our program:

```
Success = Request(Requester, Window);
   D0       -240      A0         A1
```

This call blocks the transfer of input. We only get signals from the requester. It is best if you write a subroutine for handling the requester input. We form the program very distinctly by calling the routine after initialization:

```
FileSelectBox_Request();
```

The routine itself tests the data transfer as before. We must pay special attention to the `REQCLEAR` flag. It indicates that the user has ended input and would like to return to normal program execution. The signal is released from all gadgets that set the `ENDGADGET` flag when activated. These are the boolean gadgets `Cancel` and `Ok` in our case:

```
/*************************************
 *                                   *
 * Function: Request Check           *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     12/28/1987                *
 *                                   *
 *                                   *
 *************************************/

FileSelectBox_Request()
   {
   BOOL Waiton = TRUE;
   ULONG MessageClass;

   struct IntuiMessage *message;
   struct Gadget       *GadgetPtr;
   USHORT GadgetID;

   while (Waiton)
      {
      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
         Wait(1L << FirstWindow->UserPort->mp_SigBit);
         printf("SIGNAL!\n");
         continue;
         }

      MessageClass = message->Class;
      GadgetPtr = (struct Gadget *) message->IAddress;
      GadgetID  = GadgetPtr->GadgetID;
      ReplyMsg(message);

      if (MessageClass == REQCLEAR)
         {
         Waiton = FALSE;
         printf("End Requester\n");
         continue;
         }

      if (MessageClass == GADGETUP)
      switch (GadgetID)
         {
         case SLIDER   : printf("Slider\n");
                         break;

         case PATH     : printf("Path\n");
                         break;

         case FILE     : printf("File\n");
                         break;

         case RETURN   : printf("Return!\n");
```

```
                        break;

            }

        if (GadgetID <= 16 && 1 <= GadgetID)
            {
               /* Table output */
            printf("Gadget-Nr. %d\n", GadgetID);
            }

        }
    }
```

3.5.4 Requesters and accessing the user

Double menu requesters allow more service features. With this requester type the user can call a requester whenever he wants to use its functions. The call occurs by double-clicking the right mouse button (also called the menu button).

A `DMRequest` is released every time the user double-clicks the right mouse button. This requester can be activated for a window with `SetDMRequest()`. When the requester appears, the program receives a report over the IDCMP that the first requester is active. It can then branch off to a special check loop. Here the handling is the same as for any other requester. When the program doesn't have `DMRequest` capability, everything returns to normal with `ClearDMRequest()`.

A program is especially user friendly when it offers a `DMRequest`. For example, the file select box could be made from multiple `DMRequesters`. Then the routine checks if the file should be saved, loaded or deleted, instead of letting the user select a filename then performing the function. Then each disk operation can be performed with a double-click on the right mouse key.

The requester handling for the programmer remains unchanged. The user can work with the requester in the method described above, and then read the data. The major difference occurs in the call. While conventional requesters execute from a function call in the program, this requester is only defined and can be called anytime. That's why it is important to check the `IDCMP` flags. They can test the first encounter with the requester because the previous routines bypassed the `ClearDMRequest()`.

Here is the complete file requester program:

```
/*************************************
*    3.5                             *
*   Program: Custom Requester        *
*************************************
*                                    *
* Author:   Date:    Comments        *
* ------    ----     --------        *
* Wgb      12-26-88                  *
*                                    *
* Compile and link options           *
* -----------------------            *
* cc CustomRequest (-n for SourceDB) *
* ln CustomRequest -lc +Cd (-g)      *
*                                    *
*************************************/

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

struct IntuitionBase *IntuitionBase;
struct Screen        *FirstScreen;
struct Window        *FirstWindow;
struct IntuiMessage  *message;



unsigned char PathBuffer[512] = "df1:";
unsigned char FileBuffer[31]   = "";
unsigned char UndoBuffer  [512];

USHORT KnobGraphic[] =
   {
   0xCCCC,   0x6666,   0x3333,   0x9999,
   0xCCCC,   0x6666,   0x3333,   0x9999,
   0xCCCC,   0x6666,   0x3333,   0x9999,
   0xCCCC,   0x6666,   0x3333,   0x9999
   };

struct Image Knob =
   {
   0, 0, 16, 16, 1, &KnobGraphic[0], 1, 0, NULL
   };

struct PropInfo SliderProp =
{
   FREEVERT,
   0x8000,   0x8000,
   0x0800,   0x1000,
   0,        0,
   0,        0,
   0,        0
};

struct StringInfo PathInfo =
   {
   &PathBuffer[0], &UndoBuffer[0],
```

```
    5, 511, 0, 0, 0, 0, 0, 0, NULL, 0, NULL
    };

struct StringInfo FileInfo =
    {
    &FileBuffer[0], &UndoBuffer[0],
    0, 31, 0, 0, 0, 0, 0, 0, NULL, 0, NULL
    };

SHORT StringPairs[] =
    {
    0, 0, 248, 0
    };

SHORT SelectPairs[] =
    {
    0, 0, 61, 0, 61, 21, 0, 21, 0, 0,
    };

struct Border StringBorder =
    {
    0, 9, 1, 0, JAM1, 2, StringPairs, NULL,
    };

struct Border SelectBorder =
    {
    -1, -1, 1, 0, JAM1, 5, SelectPairs, NULL,
    };

struct IntuiText PathText =
    {
    1, 0, JAM2, -58, 1, NULL, (UBYTE *)"Path:" ,NULL,
    };

struct IntuiText FileText =
    {
    1, 0, JAM2, -58, 1, NULL, (UBYTE *)"File:" ,NULL,
    };

struct IntuiText OKText =
    {
    1, 0, JAM2, 22, 5, NULL, (UBYTE *)"OK" ,NULL,
    };

struct IntuiText CancelText =
    {
    1, 0, JAM2, 2, 5, NULL, (UBYTE *)"Cancel" ,NULL,
    };

struct IntuiText ReturnText =
    {
    1, 0, JAM2, 2, 5, NULL, (UBYTE *)"CHDIR!" ,NULL,
    };

struct Gadget Files[15];
struct Gadget FileGadget =
```

```
    {
    NULL,
    80, 0, 240, 8,
    GADGHCOMP, RELVERIFY, BOOLGADGET | REQGADGET,
    NULL, NULL, NULL,
    NULL, NULL, 0, NULL
    };

struct Gadget Slider =
    {
    &Files[0],
    322, 30, 24, 120,
    GADGHNONE | GADGIMAGE, RELVERIFY, PROPGADGET |
REQGADGET,
    (APTR)&Knob, NULL, NULL,
    NULL, (APTR)&SliderProp, 1, NULL
    };

struct Gadget Path =
    {
    &Slider,
    80, 150, 250, 8,
    GADGHCOMP, RELVERIFY, STRGADGET | REQGADGET,
    (APTR)&StringBorder, NULL, &PathText,
    NULL, (APTR)&PathInfo, 21, NULL
    }; /*165 pal*/

struct Gadget File =
    {
    &Path,
    80, 165, 250, 8,
    GADGHCOMP, RELVERIFY, STRGADGET | REQGADGET,
    (APTR)&StringBorder, NULL, &FileText,
    NULL, (APTR)&FileInfo, 22, NULL
    };  /*185pal*/

struct Gadget OK =
    {
    &File,
    10, 40, 60, 20,
    GADGHCOMP, RELVERIFY | ENDGADGET , BOOLGADGET |
REQGADGET,
    (APTR)&SelectBorder, NULL, &OKText,
    NULL, NULL, 23, NULL
    };

struct Gadget Cancel =
    {
    &OK,
    10, 65, 60, 20,
    GADGHCOMP, RELVERIFY | ENDGADGET , BOOLGADGET |
REQGADGET,
    (APTR)&SelectBorder, NULL, &CancelText,
    NULL, NULL, 24, NULL
    };
```

```
struct Gadget Return =
   {
   &Cancel,
   10, 90, 60, 20,
   GADGHCOMP, RELVERIFY, BOOLGADGET | REQGADGET,
   (APTR)&SelectBorder, NULL, &ReturnText,
   NULL, NULL, 25, NULL
   };

struct Requester FileSelectBox;

struct NewWindow FirstNewWindow =
   {
   160, 0,              /* LeftEdge, TopEdge    */
   370, 200,             /* Width, Height        */
   0, 1,                 /* DetailPen, BlockPen */
   CLOSEWINDOW |         /* IDCMP Flags          */
   GADGETUP |
   REQCLEAR,
   WINDOWDEPTH |         /* Flags                */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                 /* First Gadget         */
   NULL,                 /* CheckMark            */
   (UBYTE *)"Requester Test",
   NULL,                 /* Screen               */
   NULL,                 /* BitMap               */
   100, 50,              /* Min Width, Height    */
   640, 200,             /* Max Width, Height    */
   WBENCHSCREEN,         /* Type                 */
   };

#define SLIDER 1
#define PATH  21
#define FILE    22
#define RETURN 25

main()
   {
   ULONG MessageClass;
   USHORT code;

   struct Message *GetMsg();

   Open_All();

   Make_Files(Files);


   InitRequester(&FileSelectBox);

   FileSelectBox.LeftEdge = 2;
   FileSelectBox.TopEdge  = 10;
   FileSelectBox.Width    = 360;
```

```
   FileSelectBox.Height   = 200;
   FileSelectBox.ReqGadget= &Return;

   Request(&FileSelectBox, FirstWindow);
   FileSelectBox_Request();
   EndRequest(&FileSelectBox, FirstWindow);

/*  Close_All();
  exit(TRUE);
*/
   FOREVER
      {
      if (message = (struct IntuitMessage *)
         GetMsg(FirstWindow->UserPort))
         {
         MessageClass = message->Class;
         code = message->Code;
         ReplyMsg(message);
         switch (MessageClass)
            {
            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;
            }
         }
      }
   }



/*************************************
 *                                   *
 * Function: Open library and window *
 * ================================= *
 *                                   *
 * Author:  Date:      Comment:      *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987                *
 *                                   *
 *                                   *
 *************************************/

Open_All()

   {
   void          *OpenLibrary();
   struct Window *OpenWindow();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition library not found!\n");
      Close_All();
      exit(FALSE);
      }
```

```
   if (!(FirstWindow = (struct Window *)
       OpenWindow(&FirstNewWindow)))
      {
      printf("Window will not open!\n");
      Close_All();
      exit(FALSE);
      }
   }

/*************************************
 *                                   *
 * Function: Close everything opened *
 * ================================= *
 *                                   *
 * Author:  Date:      Comment:      *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  just Intuition *
 *                     and window    *
 *                                   *
 *************************************/

Close_All()

   {
   if (FirstWindow)     CloseWindow(FirstWindow);
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }


/*************************************
*                                    *
* Function:  Gadget duplication      *
*                                    *
*************************************/

Make_Files(Struktur)

struct Gadget Struktur[];

   {
   int i;

      for (i=0; i<15; i++)
         {
         Struktur[i]             = FileGadget;
         Struktur[i].TopEdge     = 30 +i * 8;
         Struktur[i].GadgetID    = i + 1;
         Struktur[i].NextGadget  =&Struktur[i+1];
         }
      Struktur[14].NextGadget = NULL;
   }

/*************************************
 *                                   *
 * Function: Request Check           *
 * ================================= *
```

```
 *                                      *
 * Author: Date:       Comments:        *
 * ------  ----------  ----------       *
 * Wgb     12/28/1987                   *
 *                                      *
 *                                      *
 ****************************************/

FileSelectBox_Request()
   {
   BOOL Waiton = TRUE;
   ULONG MessageClass;

   struct IntuiMessage *message;
   struct Gadget       *GadgetPtr;
   USHORT GadgetID;

   while (Waiton)
      {
      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
         Wait(1L << FirstWindow->UserPort->mp_SigBit);
         printf("SIGNAL!\n");
         continue;
         }

      MessageClass = message->Class;
      GadgetPtr = (struct Gadget *) message->IAddress;
      GadgetID  = GadgetPtr->GadgetID;
      ReplyMsg(message);

      if (MessageClass == REQCLEAR)
         {
         Waiton = FALSE;
         printf("End Requester\n");
         continue;
         }

      if (MessageClass == GADGETUP)
      switch (GadgetID)
         {
         case SLIDER   : printf("Slider\n");
                         break;

         case PATH     : printf("Path\n");
                         break;

         case FILE     : printf("File\n");
                         break;

         case RETURN   : printf("Return!\n");
                         break;

         }
```

```
        if (GadgetID <= 16 && 1 <= GadgetID)
           {
              /* Table output */
           printf("Gadget-Nr. %d\n", GadgetID);
           }

        }
    }
```

3.6 Alerts

Alerts are relatives of the requester family. Alerts receive top priority from the system, but they're at the bottom of the user-friendliness list. Both requesters and alerts help get a decision from the user. Let's look at the differences between alerts and requesters:

- Requesters are highly developed windows that collect data: Alerts are screen areas that display the last information possible before the system crashes.
- You can make requesters in any size, position, and form: Alerts shove all screens aside, or append themselves to the current display. In addition, Alerts can only be displayed on half of the screen at full screen width.
- Requesters have many input options and support all gadget types: Alerts allow only two different inputs, accessed through the mouse button.
- Requesters only pause the task from which you call them: Alerts halt the entire operating system.

You now know the differences between the two communication channels. Alerts have very special uses in our programs.

Intuition manages Guru meditations as well as an `AutoRequest` in an alert when not enough memory exists. Alerts need very little memory for using the Amiga's graphic capabilities.

Note: Alerts pack a certain amount of shock value. Is the problem so serious that you want an alert to appear? Or is it something recoverable, that would be a lot friendlier if you used a requester instead? Alerts upset most users, whether the message is serious or not. Therefore, always think carefully about the proper programming of requesters and alerts, if only out of courtesy to the user.

3.6.1 Applications for alerts

The first important alert application is `Insufficient memory`. This can cause a library or screen to fail to open. We gave `Open_All()` and `Close_All()` a text using `printf()` during the test phase of our functions. But the user may not always open the CLI window and start the program using it. If this "message channel" is closed, we can report this with an alert.

This can be used in many cases if, for example an interface card must be present to operate our program, or perhaps executable memory cannot handle the program size.

The above mentioned reasons could generate an alert if you don't think that a simple `AutoRequester` will be enough. The alert stops the execution of the entire system and waits for you to react. This alert places all other processes in a lower priority.

3.6.2 Creating and configuring an alert

The `DisplayAlert()` function needs three arguments to construct an alert: Alert type, alert height and alert text.

Alert type We first assign the number which Intuition reads and handles as the alert type. You differentiate between two recognizable types by their most significant byte of the `long` variable. A `RECOVERY_ALERT` (which returns to the program after the alert display) has a label of 00, and a `DEADEND_ALERT` (displays the alert then resets the machine) has a label of 80.

Alert height We give the function the height of our alert in screen lines. The alert then pushes all screen lines below it to fit on the top of the screen.

Alert text The pointer to the alert text points to one string that states the desired information. The alert text consists of many alert strings concluded by continuation bytes. Continuation bytes are either end of line characters or end of paragraph characters. At the beginning each alert string has a position statement in pixels, followed by the text in normal string format. Then we find a byte that reads the next string, if it is unequal to null. The alert text closes when done loading the strings.

Unfortunately, Intuition supports little of the complications of alert management. You won't find the structure in the `include` file.

First we establish a structure for the alert strings:

```
/* 3.6.2.a. alertmessage_struct */
struct AlertMessage
{
0x00  00  SHORT LeftEdge;
0x02  02  BYTE TopEdge;
0x03  03  char AlertText[50];
0x53  83  BYTE Flag;
0x54  84
};
```

The biggest problem encountered when defining the structure is the length of the text. We must accept a default value for this; you can change it to anything. Be careful that the alert appears in `topaz sixty` font, and that the text line is no longer than 60 characters.

We can now lump our alert strings together into one unit. For this we establish an array in this structure:

```
/* 3.6.2.b. firstalert */
#define NOEND 0xFF
#define END   0x00

struct AlertMessage UserAlert[] =
{
   50, 24, " This is the first alert I have written!!", NOEND,
   50, 34, "  ----------------------------------- ", NOEND,
   50, 44, "                                      ", NOEND,
   50, 54, " <Left Button>  CANCEL  <Right Button>  RETRY", END
};
```

We defined two lables to signal the continuation bytes. NOEND indicates that another text follows, and END indicates the end of the text.

After defining our first alert structure, we can call it with the `DisplayAlert()` function. This function has the following format:

```
Resply   = DisplayAlert(AlertNumber, String, Height);
   D0          -90          D0          A0      A1
```

We can insert the following known values into our text program:

```
Reply = DisplayAlert(RECOVERY_ALERT, &UserAlert, 50L);
```

Here is the entire source code for an alert:

```
/*************************************
 *                                   *
 * Program: First Alert              *
 * ================================= *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     12/29/1987  First Alert   *
 *                                   *
 *************************************/
```

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


struct IntuitionBase *IntuitionBase;

struct AlertMessage
{
   SHORT LeftEdge;
   BYTE TopEdge;
   char AlertText[50];
   BYTE Flag;
};

#define NOEND 0xFF
#define END   0x00

struct AlertMessage UserAlert[] =
{
   50, 24, "    This is the first alert I have written!!!
", NOEND,
   50, 34, "    ----------------------------------------
", NOEND,
   50, 44, "
", NOEND,
   50, 54, "    <Left Button> CANCEL   <Right Button>
RETRY", END
};

main()
   {
   BOOL Reply;

   Open_All();

   Reply = DisplayAlert(RECOVERY_ALERT, &UserAlert, 80L);

   if (Reply == TRUE)
      printf("Left mouse button was pressed!\n");
   else
      printf("Right mouse button was pressed!\n");

   Close_All();
   }


/*************************************
 *                                   *
 * Function: Open all                *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  for Intuition *
 *                                   *
 *************************************/
```

```
Open_All()
   {
   void          *OpenLibrary();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }
   }


/*************************************
 *                                   *
 * Function: Cloae all               *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  for Intuition *
 *                                   *
 *************************************/

Close_All()
   {
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

How it works The program first prepares all of the necessary structures for the function call. Both `include` files are included because without their definitions, the alert is not possible.

If you define the alert string with all of the coordinates and constants, the main program can call the `Open_All()` function to ensure access to Intuition. Then `DisplayAlert()` starts and processes the return value. Everything can be closed again with `Close_All()`.

Note: To examine the workings of a `DEADEND_ALERT`, you should change the alert type. Remember that in a `DEADEND_ALERT`, control passes on to the reset routine.

3.6.3 Alerts for program projects

An alert built into a program means work for the programmer that could have been spared. The aim in programming is to keep the process as efficient as possible.

It's important to install additional alerts at the end of all work in the program. These alerts usually replace error messages given in the CLI window during the development phases.

We recommend the following system for appending alerts: Your program is ready and you've already determined all points in the program at which an alert can replace a simple error message. Now you can develop the text to be displayed in the proper case.

For this we recommend a general alert design, and maybe a multi-alert. Multi-alerts insert just a specification on the margin. The alert texts can be stored in a file accessed from the structure definition. This file is included last:

```
#include <exec/devices.h>
#include <graphics/gfxbase.h>

...

#includes "Disk:Directory/Alert"
```

In the `Alert` file we find, for example, the following error messages for a word processor:

```
/*****************************************/
/*                                       */
/* 3.6.3.a. alertinclude                 */
/*                                       */
/* include : Alert Structure w/ Definition */
/* ===================================== */
/*                                       */
/* Author:    Date:     comments:        */
/* ------  ----------  ----------        */
/* Wgb     29.12.1987  for word processor */
/*                                       */
/*****************************************/


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


struct AlertMessage
{
   SHORT LeftEdge;
   BYTE TopEdge;
   char AlertText[50];
   BYTE Flag;
};

#define NOEND 0xFF
#define END   0x00
```

```
struct AlertMessage OutMem[] =
{
  50, 24, "  There is not enough memory          ", NOEND,
  50, 34, "  Please free up some memory!         ", NOEND,
  50, 44, "  Start the program again!            ", NOEND,
  50, 54, "                                      ", NOEND,
  50, 64, "     Mouse key to continue            ", END
};

#define OutMemSize 80L


struct AlertMessage NoLibrary[] =
{
  50, 24, "  Graphics.library cannot be opened  ", NOEND,
  50, 34, "    The program is stopped!          ", NOEND,
  50, 44, "  Make sure that this library is     ", NOEND,
  50, 54, "  on the Workbench!                  ", NOEND,
  50, 64, "                                     ", NOEND,
  50, 74, "       Mouse key to continue         ", END
};

#define NoLibraySize 90L


struct AlertMessage Ende[] =
{
  50, 24, "   Important functions cannot be     ", NOEND,
  50, 34, "   reached!                          ", NOEND,
  50, 44, "   Crash!  Crash! .....              ", NOEND,
  50, 54, "                                     ", NOEND,
  50, 64, "      Mouse key to continue          ", END
};

#define EndeSize 80L
```

For the `DisplayAlert()` call we included the height needed for each alert in a `#define`. The program text contains the following lines:

```
if (Window == NULL)
   {
   printf("Unfortunately no window can be opened!\n");
   Close_All();
   exit(FALSE);
   }
```

Replace it with this:

```
if (Window == NULL)
   {
   DisplayAlert(RECOVERY_ALERT, &NoWindow, NoWindowSize);
   Close_All();
   exit(FALSE);
   }
```

3.6.4 Understanding the Guru Meditation

Even if you've never programmed any alerts, you probably have had an acquaintance with Guru Meditations. Generally you assume the message on the top line is true because most folks can't interpret the funny numbers underneath. This is a misunderstanding: Gurus report to the user why the operating system must be reset.

Gurus and why This is especially important if you do your own programming. If you know why your own program runs into errors in the test phase, you can usually find the source of the errors. The only problem with the Guru Meditation is that it always seems to display the same text, and a number which doesn't give a clue as to the problem. This is something we can help change. The following tables are used to decipher any Guru Meditation.

Guru numbers The Guru number consists of multiple segments. We can break them down into a general format:

```
Guru Meditation : TTSSFFGGG.AAAAAAAA
```

The letters have the following meanings:

Letters in code	Description
TT	ALERT_TYPE
SS	System class
FF	Error class
GGGG	Exact description
AAAAAAAA	Address of error solving task

Just a word about the Guru codes listed below: We write out the entire term along with the number code. If an alternate exists, we include the word OR.

There are only two possibilities for TT. You read about these values under the label ALERT_TYPE. It can be either a DEADEND_ALERT (80) or a RECOVERY_ALERT (00) (only the latter type returns data).

ALERT_TYPES

00000000	RECOVERY ALERT
80000000	DEADEND ALERT

Next we will discuss the system class. Here we find the SUBSYSTEM_CODE label, where the error occurs.

SUBSYSTEM_CODE

00000000	68000

LIBRARIES

01000000	exec.library
02000000	graphics.library
03000000	layers.library
04000000	intuition.library
05000000	math.library
06000000	clist.library
07000000	dos.library
08000000	ram.library
09000000	icon.library
0A000000	expansion.library

DEVICES

10000000	audio.device
11000000	console.device
12000000	gameport.device
13000000	keyboard.device
14000000	trackdisk.device
15000000	timer.device

RESOURCES

20000000	CIA
21000000	Disk
22000000	Misc

MISC

30000000	Bootstrap
31000000	Workbench
32000000	DiskCopy

Be sure that the error message is at the beginning of the subsystem list. Here the processor automatically gets the words. When this occurs, it acts as a processor trap. The rest of the locations of the Guru Meditation have other meanings:

TRAP_CODES

00000002	Data or address bus error with measures
00000003	Addressing error (uneven address)
00000004	Illegal instruction
00000005	Division by zero
00000006	CHR Instruction
00000007	TRAPV Instruction
00000008	Privilege violation
00000009	Single-step mode
0000000A	Line A emulator (opcode 1010)
0000000B	Line F emulator (opcode 1111)

If an error occurred after the subsystem list, then the ERROR_CLASS code brings us closer to the cause of the error:

ERROR_CLASS

00010000	Not enough memory
00020000	Library cannot be constructed
00030000	Library cannot be opened
00040000	Device cannot be opened
00050000	No reaction from the hardware
00060000	I/O error
00070000	I/O not present

The last four numbers of the error code give us still more detailed information. These are specified in the subsystems:

exec.library

01000000	checksum error excepting CPU
81000002	checksum error at the starting address
81000003	checksum of a library
81000004	not enough memory for library
81000005	error is memory list entry
81000006	not enough memory for interrupt
81000007	pointer error
81000008	error in Semaphore
81000009	memory area was freed up for the second time
8100000A	pointer error with exception

graphics.library

82010001	not enough memory for copper list
82010002	not enough memory for copper instruction list
82010003	the copper list is full
82010004	division error in the copper list
82010005	not enough memory for the copper list head
82010006	not enough memory for "long frame"
82010007	not enough memory for "short frame"
82010008	not enough memory for the fill routine
82010009	not enough memory for the text routine
8201000A	not enough memory for the blitter bit-map
8201000B	incorrect memory area
82010030	error during the establishment of a view port
82011234	GfxNoLCM (no intermediate memory available)

layers.library

03000001	not enough memory for Layers

intuition.library

84000000	gadget type is unknown
04000001	type error with AN_gadget
84010002	not enough memory for building a port
04010003	not enough memory for a menu
04010004	not enough memory for a submenu
84010005	not enough memory for the menu list
84000006	incorrect position of the menu list
84010007	not enough memory for OpenScreen()
84010008	not enough memory for building a RastPort
84000009	unknown or erroneous SCREEN_TYPE
8401000A	not enough memory for gadget
8401000B	not enough memory for window
8400000C	bad register status when intuition.library open
8400000D	incorrect message over the IDCMP
8400000E	not enough memory for the message stack
8400000F	not enough memory for the console.device

dos.library

07000001	not enough memory at start up
07000002	the task was not ended
07000003	Qpkt error is encountered
07000004	data packet was not expected
07000005	free pointer is inaccessible
07000006	erroneous data of a disk block
07000007	the bit-map is destroyed
07000008	key was already freed up
07000009	the check sum is full of errors
0700000A	disk error
0700000B	key is outside of the allowable area
0700000C	incorrect over write

ram.library

08000001	erroneous entry in the management list

expansion.library

0A000001	error in the expansion hardware

trackdisk.device

14000001	error during the search
14000002	error with the timer impulse: delay

timer.device

15000001	error with access of the device
15000002	error with time coordination: network fluctuation

disk.resource

21000001	inserted disk was unknown
21000002	interruption because no drive is connected

bootstrap

30000001	error when analyzing the boot data

3.7 Checking the IDCMP

A program really "comes to life" when it interacts with its user. We're going to write a small demonstration program that asks for user input. The big question: How to get information from the user?

The answer: The Amiga has many input and output devices for data. These include the `audio.device`, `console.device`, `timer.device`, `trackdisk.device`, `input.device`, `keyboard.device`, `gameport.device`, `narrator.device`, `serial.device`, `parallel.device`, `printer.device`, and the `clipboard.device`. Some of these devices only provide input, others only provide output, while some can perform both tasks. For example, we can only transmit data over the `audio.device`, `narrator.device` and `printer.device`. We can only receive data from the `timer.device`, `keyboard.device` and `gameport.device`. Data can be transferred in both directions with the remaining devices. That's good communication.

All of these input and output devices have a disadvantage. When we want to send and receive multiple data of different types, we need more than one device to perform the task. Many devices have certain advantages, while others have strong disadvantages. The end result is that the program opens a number of devices, but the programmer may lose sight of what's open and what isn't.

For this reason the Amiga's developers came up with the `IDCMP` (Intuition Direct Communications Message Port). This is a combination of `input.device`, `gameport.device`, `timer.device`, `trackdisk.device` and `console.device`, with interfacing through Intuition.

The `IDCMP` is for data processing, filtering and preparation. Through the assigned window you can specify which data you want to receive and which data it should convert into the proper format. Or maybe you would prefer to leave the data unchanged? This is also possible with the `IDCMP`. If you would like to have more flexibility, then you should avoid the `graphics.library`. Intuition is usually enough to handle this. The `IDCMP` receives all input, and the graphic structures make output possible.

Let's take a closer look at the `IDCMP` and its versatile input capabilities.

3.7.1 Configuration and receiving

The IDCMP is not constantly on standby, ready for the programmer's call. Intuition must know that a possibility exists for data reception.

First we need a window. Without a window, the Amiga can't receive data through Intuition. Remember the NewWindow structure. The IDCMPFlags entry lists the interface to IDCMP. When a flag indicates that we are interested in the information, Intuition establishes a port for this window. We must first know which flags can be set. These are divided into six groups.

The first group lists the most important flags. They occupy the message port window:

SIZEVERIFY When this flag is set the program receives a message if the user changes the size of the window. This can be important if the program is drawing and this work must stop before the size can be changed. Thus the window size cannot be changed before verification.

NEWSIZE When this flag is set the program receives a message of a size change in the window. The value can be read from the Window structure.

REFRESHWINDOW
When this flag is set the program receives a message of whether the window contents must be redrawn. Used on SIMPLE_REFRESH and SMART_REFRESH window types. SMART_REFRESH windows only need the refresh flag when they are supplied with a sizing gadget.

ACTIVEWINDOW
When this flag is set the program receives a message when the window is active.

INACTIVEWINDOW
When this flag is set the program receives a message when another window is activate.

The second group concentrates on the problem of gadget manipulation. We learned about these flags earlier with our gadget check:

GADGETDOWN When this flag is set the program receives a message when a gadget of type GADGIMMEDIATE is clicked on.

GADGETUP When this flag is set the program receives a message when a gadget of type RELVERIFY is clicked on.

CLOSEWINDOW When this flag is set the program receives a message when the close gadget is activated by the user. Intuition does not close the window automatically.

The third group consists of expanded flags to control the gadgets.

REQSET When this flag is set the program gets a message when the user clicks on the first requester in the window.

REQCLEAR When this flag is set the program gets a message when the user clicks on the last requester in the window.

REQVERIFY When this flag is set the program gets a message when Intuition tries to open the first requester. Intuition then waits for a response to the message. You can finish all output before the requester appears. If further requesters open in the window, Intuition does not give a message. During requester handling the output remains visible.

The fourth group handles menu reading. Menu programming will be described in the next section.

MENUPICK When this flag is set the program gets a message when the user presses the right mouse (menu) button.

MENUVERIFY When this flag is set the program gets a message when someone tries to activate the menus. As long as no response occurs (e.g., with `Reply()`) the menus cannot be presented.

The flags of the fifth group check mouse status:

MOUSEBUTTONS When this flag is set the program gets a message whether a mouse button is pressed or not. Remember that no messages are sent when the left button is pressed while the pointer is over a gadget, or when the right button is used to manipulate the menus.

MOUSEMOVE When this flag is set the program gets a message when the user moves the mouse. The REPORTMOUSE flag must also be set (or the FOLLOWMOUSE flag, when a gadget is involved).

DELTAMOVE When this flag is set the program gets a message when the user moves the mouse. Unlike the above

flag, the position returned is the difference from the last position.

In the sixth group we find the unassigned flags.

`INTUITICKS` When this flag is set the program gets a message when a tenth of a second elapses (in conjunction with the `timer.device`). You only get further `TICK` reports if the last one got a response.

`NEWPREFS` When this flag is set the program gets a message when Preferences values change. This is important, for example, if the program abandons the color settings or the printer setting. Every window in which this flag is set receives a message. A program that changes the Preferences data checks to see if it wants to inform the others.

`DISKINSERTED`
When this flag is set the program gets a message when a disk is inserted into the drive. Every window in which this flag is set gets a message (in conjunction with the `trackdisk.device`).

`DISKREMOVED` When this flag is set the program gets a message when a disk is removed from a drive. Every window in which this flag is set gets this message.

`RAWKEY` When this flag is set the program gets a message when a key on the keyboard is pressed. The keyboard code is received in "raw" form, which means that code doesn't get converted according to the keyboard table (in conjunction with the `input.device`).

`VANILLAKEY` When this flag is set the program gets a message when a key is pressed. The keyboard code is processed and converted according to the keyboard table (in conjunction with the `console.device`).

Here are all of the flags and their hex values:

Flag name	Hex value	Remarks
SIZEVERIFY	0x00000001L	Window flags
NEWSIZE	0x00000002L	
REFRESHWINDOW	0x00000004L	
ACTIVEWINDOW	0x00040000L	
INACTIVEWINDOW	0x00080000L	
GADGETDOWN	0x00000020L	Gadget flags
GADGETUP	0x00000040L	
CLOSEWINDOW	0x00000200L	
REQSET	0x00000080L	Requester flags
REQCLEAR	0x00001000L	
REQVERIFY	0x00000800L	
MENUPICK	0x00000100L	Menu flags
MENUVERIFY	0x00002000L	
MOUSEBUTTONS	0x00000008L	Mouse flags
MOUSEMOVE	0x00000010L	
DELTAMOVE	0x00100000L	
INTUITICKS	0x00400000L	timer.device
NEWPREFS	0x00004000L	Preferences change
DISKINSERTED	0x00008000L	trackdisk.device
DISKREMOVED	0x00010000L	
RAWKEY	0x00000400L	input.device
VANILLAKEY	0x00200000L	console.device
WBENCHMESSAGE	0x00020000L	Workbench message
LONELYMESSAGE	0x80000000L	No Intuition/IDCMP message

There are a number of ways to tell the system which region you decided should receive the messages and information. The first and the simplest: Insert the corresponding flags in the NewWindow structure. Intuition establishes the IDCMP; you need only worry about the check. The second way: Use the ModifyIDCMP() function to display a second window for information display, without establishing a port. Simply give the window pointer and all of the flags you want set. This either establishes a new port or closes an existing one. The command has the following format:

```
ModifyIDCMP(Window, IDCMPFlags);
   -150       A0        D0
```

The third possibility converts the previous flags to the new flags' values.

The functions that influence the IDCMP, such as REPORTMOUSE(), are also important for the second method. Here the same thing happens, but in a single flag.

3.7.1.1 The IntuiMessage structure

Now that you have some background on the situations for supplying the IDCMP messages, it's time to look at exactly what gets reported in transmission.

We find in the Window structure a pointer to the Intuition structure named MessageKey at byte 90. We also find another pointer at byte 82 which points to the MessagePorts named UserPort and WindowPort. You can process the input in conjunction with these three message ports.

First let's look at the IntuiMessage structure:

```
/* 3.7.1.1.a.intuimessage_struct */
struct IntuiMessage
{
0x00  00  struct Message ExecMessage;
      0x00  00  struct Node mn_Node;
            0x00  00  struct Node *ln_Succ;
            0x04  04  struct Node *ln_Pred;
            0x08  08  UBYTE ln_Type;
            0x09  09  BYTE ln_Pri;
            0x0A  10  char *ln_Name;
            0x0E  14
      0x0E  14  struct MsgPort *mn_ReplyPort;
      0x12  18  UWORD mn_Length;
      0x14  20
0x14  20  ULONG Class;
0x18  24  USHORT Code;
0x1A  26  USHORT Qualifier;
0x1C  28  APTR IAddress;
0x20  32  SHORT MouseX;
0x22  34  SHORT MouseY;
0x24  36  ULONG Seconds;
0x28  40  ULONG Micros;
0x2C  44  struct Window *IDCMPWindow;
0x30  48  struct IntuiMessage *SpecialLink;
0x34  52
};
```

Structure description

The structure begins with a message structure. This is needed for the data transfer of messages from Exec. This is interesting and that additional information can contain important data for other purposes. The other information depends on the flag.

Variables

Exec needs ExecMessage so it can transfer the message into the system.

Class is a ulong variable that marks the type of the data sent. These bits match the window IDCMP variables, and may be necessary to identify the message.

Code contains data which depends on the released flag. The value always has different meanings.

Qualifier is a copy of the ie_Qualifier variables transferred from input.device.

MouseX and MouseY contain the mouse coordinates of the window up to the time of the message transmission.

Seconds, Micros duplicate the Amiga's system clock of the Amiga up to the time of the message transmission.

IAddress is a pointer to the structure where you find the report. That can, for example, be a gadget structure from which the impulse was released.

IDCMPWindow is a pointer to the window from which the message stemmed.

SpecialLink is a variable that is used from the system.

As you see, the IntuiMessage structure holds all of the data that can be important for further processing. Here we find the basic data of Intuition, whose evaluation is dependent on the impulse received.

3.7.1.2 MsgPort structures

The MsgPort structure named UserPort signals the receipt of the message. UserPort doesn't lie within the IntuiMessage structure. By setting a bit in the variable mp_SigBit we can wait for a message. The structure contains the following elements:

```
/* 3.7.1.2.a. msgport struct */
struct MsgPort
{
0x00  00  struct Node mp_Node;
    0x00  00  struct Node *ln_Succ;
    0x04  04  struct Node *ln_Pred;
    0x08  08  UBYTE ln_Type;
    0x09  09  BYTE ln_Pri;
    0x0A  10  char *ln_Name;
    0x0E  14
0x0E  14  UBYTE mp_Flags;
0x0F  15  UBYTE mp_SigBit;
0x10  16  struct Task *mp_SigTask;
0x14  20  struct List mp_MsgList;
0x22  34
};
```

Structure description

The structure begins with a node structure which helps in recognizing the port.

`mp_Flags` labels the type of message:

`PA_SIGNAL`	is set when a message is expected
`PA_SOFTINT`	releases a software interrupt
`PA_IGNORE`	labels message that should be ignored

`mp_SigBit` is the variable that relates to us. The set bit indicates the arrival of a task signal for us.

`mp_SigTask` is a pointer to the task that sent the signal.

`mp_MsgList` heads the list of all of the messages sent to this port.

You may be confused after reading all this, considering all we want to do is check the `IDCMP`. The `IDCMP`'s complexity stems from its close relationship with the Amiga's internal signal system. All we need to understand for now is the `mp_SigBit` of the `IDCMP`. We wait for a message through the signal for this bit. What can be found in this bit is immaterial.

3.7.1.3 Waiting for a message

Intuition establishes a message port and an `IntuiMessage` structure for us. Our program should wait until it receives a message that it can read and evaluate. The best route would be to use the `Exec` function `GetMsg()` to see if `UserPort` contains a message:

```
Message = GetMsg(Window->UserPort);
```

We receive one of two results: Either `GetMsg()` returns a pointer to the message structure (in this case, an `IntuiMessage` structure), or a null pointer (i.e., no message present).

You can repeat this check using a loop until a message is present. That loop looks like this:

```
/*3.7.1.3.a.msgwait */
FOREVER
   {
   if (message = (struct IntuiMessage *)
       GetMsg(FirstWindow->UserPort))
      {
         /* Evaluation */
      }
   }
```

This message check is very compatible with our multitasking computer. Although it has nothing to do yet, our program constantly strives to read a message.

It would be better if the task were in "quiet" status until a message actually is sent. For this there is an `Exec` function named `Wait()`. `Wait()` lets you inform the system to not process the task until a message for it appears. The new version of the `FOREVER` loop looks like this:

```
/* 3.7.1.3.b.msgquiet */
FOREVER
   {
   if ((message = (struct IntuiMessage *)
       GetMsg(FirstWindow->UserPort)) == NULL)
      {
      Wait(1L << FirstWindow->UserPort->mp_SigBit);
      continue;
      }

      /* Evaluation */

   }
```

First the loop inspects the `UserPort` for a message. If no message exists, the system waits until one is received. Without a message, program execution continues at the beginning of the `FOREVER` loop. If a message does exist, or if the system returns to the `Wait` status, it reads the message. When in `Wait` status, the loop begins where the received message is read in (this is true in every case).

3.7.2 Reading the message

The message reading depends completely on the type of message received. We need to read gadgets, menus, requesters, the keyboard and the mouse. This strict subdivision increases the size and complexity of the programming, but also increases the final program's flexibility. The C language supports this flexibility, that makes programming the checks to a minimum, providing you understand the programming involved.

When receipt of the message is confirmed, the program goes to the pre-evaluation process. We must first establish the category into which the message falls (i.e., which flag value the `Class` variable of the `IntuiMessage` structure receives). For this examination we get the standard values from the structure:

```
MessageClass = Message->Class;
MessageCode = Message0>Code;
```

We can then determine from `MessageClass` which `IDCMP` flag has released the message. In this sense we must continue the inspection. Do not forget to reply to the message so the message system can continue operation. No further messages can be received without a reply:

```
ReplyMsg(Message);
```

We now create checks for individual flag groups, with which we can manage the data flow.

3.7.2.1 Gadgets

Once you set the pointer to the `IntuiMessage` structure, we can begin with the reading. Let's look at how you program this reading to accept gadget data. You have already read about the development of gadget checking.

We first test if it is handled as the gadget `IDCMP` flag:

```
if (MessageClass & (GADGETDOWN | GADGETUP))
     {
          /* Gadget check */
     }
```

Then we provide the pointer to the `Gadget` structure:

```
struct Gadget *GadgetPrt;
     GadgetPtr = (struct Gadget *) Message->IAddress;
```

For the difference between multiple gadgets we need the `GadgetId`:

```
USHORT GadgetID;
     GadgetID = GadgetPrt->GadgetID;
```

Note: We recommend successive numbers replaced by a `#define` at the beginning of the program, to allow easier recognition of gadgets.

After we determine the `GadgetID` we can release the individual actions. The source code makes more sense if you place longer reactions in a function:

```
/* 3.7.2.1.a.gadcheck */
switch(GadgetID)
   {
   case TEXTGADGET : Text_Evaluate();
                     break;

   case SLIDER     : New_Value();
                     break;

   case ....

   }
```

All further information can come through the pointer to the `Gadget` structure. The mouse coordinates can be found in the `IntuiMessage` structure—the gadgets are no longer stored here. The mouse coordinate check is very simple:

```
MouseX = (SHORT) Message->MouseX;
MouseY = (SHORT) Message->MouseY;
```

3.7.2.2 Window messages

Window flag reading controls more than gadgets. Window flags also return additional status information.

Two possibilities for this check: the first is direct insertion into the system. We first select all window flags and read them further:

```
/* 3.7.2.2.a.windowmsg */
if (MessageClass & (NEWSIZE | REFRESHWINDOW | ACTIVATEWINDOW |
                                        INACTIVATEWINDOW))
   {
   WindowPtr = (struct Window *) Message->IAddress;
   switch(MessageClass)
      {
```

```
        case NEWSIZE          : TWidth = WindowPrt->Width;
                                THeight  = WindowPrt->Height;
                                break;

        case REFRESHWINDOW    :
                                break;

        case ACTIVATEWINDOW   :
                                break;

        case INACTIVATEWINDOW: ActivateWindow(WindowPtr);
                                break;

    }
```

Bear in mind that this method cannot check the `SIZEVERIFY` because the exact inspection was answered with `ReplyMsg()` through the message. Here a simple flag check before each reply will be enough. You can simply jump to the `ReplyMsg()` function.

The last line is quite clever. These lines constantly keep the window in our program active. As soon as the user clicks on another window, the program reactivates our program window.

The Workbench also uses this method. When you select the `Rename` item from the Workbench menu, a window containing a string gadget opens. The window is the same size as the string gadget, and has no system graphics. Click anywhere on the Workbench and the `Rename` window remains active.

3.7.2.3 Requesters and execution

Requesters can normally only be exited from the program, and do not usually obstruct program execution. It gives the user some comfort that a certain function can be called at any time through `DMRequest`. However, that leaves interruption time undetermined. This is why requester flags exist.

When you place `REQSET` and `REQCLEAR` in the `IDCMP` of the window, you always receive a message, if the first requester is activated (our `DMRequest`) and when the last requester disappears (also our `DMRequest`).

Because the `End` flag receives priority over the requester routine, we only need to worry about `REQSET`. One program line circumvents this problem:

```
if (MessageClass & REQSET) DMRequest_Evaluation();
```

You can test to see if the process released or if the important work must still be executed using REQVERIFY. REQVERIFY works just like SIZEVERIFY. When the flag is set, the program first receives this message before Intuition opens the requester. This can first be processed after the message is answered with ReplyMsg().

3.7.2.4 Menu flag analysis

Menus are the most important input option offered to the user by Intuition (next to gadgets). When the user presses and releases the right mouse button, the program gets a message over the IDCMP if the MENUPICK flag is set. We can then search in the Code array of the IntuiMessage structure for the menu item number. The menu points are not managed by numbers: An internal system makes the identification process as easy as possible. The system allows us to program submenus only to a certain extent for this reason.

Now we should look at the example menu's coding. The two bytes can be "divided" as follows:

```
UUUUU MMM|MMM NNNNN
```

Here we see two bytes within three bit groups. The first group (NNNNN) describes the menu number. We must separate this from the others to get the number. The work executes the definition MENUNUM(Code), found in the <intuition/intuition.h> include file.

The second group (MMMMMM) represents the number of the menu item. It allows 31 menu titles and 63 menu items. To determine the number of menu items available, use the definition ITEMNUM(Code).

The third group (UUUUU) contains the number of the submenu item if one is present. The system allows 31 possible values here. These values can be reached through the definition SUBITEM(Code).

Definitions makes the menu checking very easy. In our check loop for all of the IDCMP messages, we must first determine whether a menu message exists. Then we can branch off to a function that can make a further analysis:

```
if (MessageClass & MENUPICK) Menu_Evaluation(Code);
```

The function calculates and utilizes the values one after another:

```
/* 3.7.2.4.a.menufigure */
Menu_Analysis(Menunumber)
USHORT Menunumber;
{
   USHORT Menu, MenuItem, SubItem;

   Menu     = MENUNUM(Menunumber);
   MenuItem = ITEMNUM(Menunumber);
   SubItem  = SUBITEM(Menunumber);

   switch(Menu)
   {
      case FILEMENU : FileAnalysis(MenuItem, SubItem);
                      break;

      case WORKMENU : WorkAnalysis(MenuItem);
                      break;
   }
}

FileAnalysis(MenuItem, SubItem)
USHORT MenuItem, SubItem;
{
   switch(MenuItem)
   {
      case LOAD     : Load();
                      break;

      case SAVE     : Save();
                      break;

      case DELETE   : switch(SubItem)
                      {
                         case FILE : DelFile();
                                     break;

                         case TEXT : DelText();
                                     break;
                      }
                      break;
   }

WorkAnalysis(MenuItem)
USHORT MenuItem;
{
   switch(MenuItem)
   {
      case MARK     : Mark();
                      break;

      case EDIT     : Edit();
                      break;

      case SEARCH   : Search();
                      break;
   }
```

Instead of numbers we used expressions for the menus, menu items and submenu items. This makes the program text easier to understand. Here are the `defines` for the above routines:

```
/* 3.7.2.4.b.menu names */
/* Menu names */

#define FILEMENU
#define WORKMENU

/* 1. Menu  FILEMENU */

#define LOAD
#define SAVE
#define DELETE

/* Sub menu DELETE */

#define FILE
#define TEXT

/* 2. Menu WORKMENU */

#define MARK
#define EDIT
#define SEARCH
```

3.7.2.5 Mouse reading

The mouse serves a number of purposes, including gadget selection and menu item selection. The programmer can also monitor the mouse's activities. The `IDCMP` directly reads the mouse. By setting flags we indicate whether we would like to know that the user pressed or released mouse buttons, or even that the user moved the mouse. Everything can be received through the `IDCMP`.

In a drawing program it can be important to know when the mouse button was pressed (e.g., so a point can be placed at that location). There are two mouse buttons, and the system can check for either button pressed or released. Accordingly there are four possible results:

`SELECTDOWN, SELECTUP`	Left mouse button (select) flags.
`MENUDOWN, MENUUP`	Right mouse button (menu) flags.

Be careful that the left mouse button only sends signals if it is not needed for gadget work (gadget operation has a high priority). The same goes for the right mouse button: The right mouse button registers only if the flag `RMBTRAP` is set.

The check looks very simple in the program. We assume that the program needs both mouse buttons and manages no windows. We set the IDCMP flags MOUSEBUTTONS and RMBTRAP in the window. When we get a message, the Code array of the IntuiMessage structure indicates the type of mouse button:

```
MousX = (SHORT) Message->MouseX;
MousY = (SHORT) Message->MouseY;

if (MessageClass & MOUSEBUTTONS) MausAbfrage(Code, MousX,
MousY);
```

These lines branch to a subroutine where further readings take place:

```
MousX = (SHORT) Message->MouseX;
MousY = (SHORT) Message->MouseY;

if (MessageClass & MOUSEBUTTONS) MouseRead(Code, MousX,
MousY);
```

This line branches to a subroutine where further readings take place:

```
/* 3.7.2.5.a.readmouse */
MouseRead(Flag, MousX, MousY)
USHORT Flag;
SHORT MousX, MousY;
{
   switch(Flag)
   {
      case SELECTDOWN : Begin_Line(MousX, MousY);
                        break;

      case SELECTUP   : End_Line(MousX, MousY);
                        break;

      case MENUDOWN   : Begin_Erase(MousX, MousY);
                        break;

      case MENUUP     : End_Erase(MousX, MousY);
                        break;
   }
}
```

The example function receives the mouse status along with the coordinates as the assignment value. Depending on which mouse status has priority, either the lines function of the program executes or ends or the delete function executes. We then assign the first or second corner pixel.

The next important point of the mouse check is the position. In the first example we used this in part because the mouse position is supplied to every IntuiMessage relative to the window. When we want to program a permanent coordinate display, we cannot wait for the user to press a key to see the coordinates. He must always have the

current condition. Intuition offers a message type for this. Simply set the IDCMP flag REPORTMOUSE, and you receive each movement.

Note: We would like to warn you at this time! Each small movement of the mouse, even one pixel, releases a message. You should also consider that many steps are needed for the exact positioning.

We recommend a check for mouse movement:

```
MousX = (SHORT) Message->MouseX;
MousY = (SHORT) Message->MouseY;
```

The subroutine can then handle analysis. With the above mentioned methods you get the absolute value for every mouse movement, relative to the IDCMP window. It is easier in many applications if you know the difference from the last position. For this change we must set the DELTAMOVE flag.

3.7.2.6 Recognizing keyboard input

Keyboard input can only be recognized when either the VANILLAKEY or RAWKEY flag is set. Either flag sets the keyboard mode we want "see." Only one mode per window is allowed.

RAWKEY is different from VANILLAKEY in that we receive untranslated codes from RAWKEY (i.e., each key sends a number as the message).

When VANILLAKEY is active, the keyboard number is translated according to the keyboard table. This gives us the keyboard table codes.

Remember this when programming, since the computer's keyboard configuration does change from country to country. In America we use an American keyboard arrangement, in Germany a German configuration, in France a French configuration, etc.

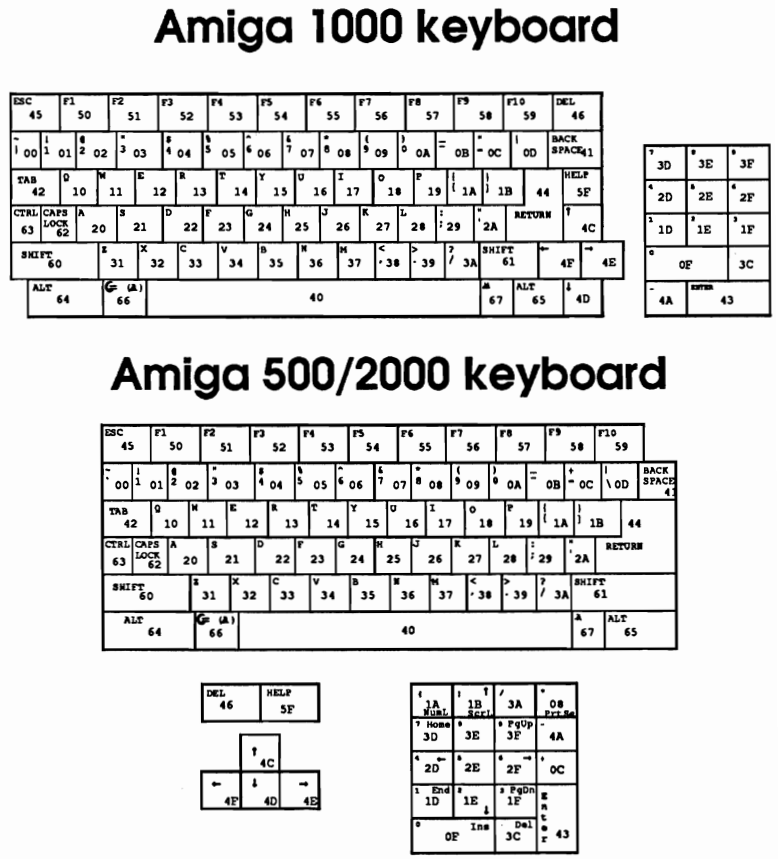


Figure 3.12
Raw Keyboard codes

Note that the numeric keypad has its own numbering system. When you want to program a check, you must search the `Class` array for the `RAWKEY` flag, and in the `Code` array for the key number. The `Qualifier` array returns a status message concerning special keys like <Shift>, <Alt> and <Ctrl>.

You can identify the key pressed with the following table:

QUALIFIER TYPES	Hex value
IEQUALIFIER_LSHIFT	0x0001L
IEQUALIFIER_RSHIFT	0x0002L
IEQUALIFIER_CAPSLOCK	0x0004L
IEQUALIFIER_CONTROL	0x0008L
IEQUALIFIER_LALT	0x0010L
IEQUALIFIER_RALT	0x0020L
IEQUALIFIER_LCOMMAND	0x0040L
IEQUALIFIER_RCOMMAND	0x0080L
IEQUALIFIER_NUMERICPAD	0x0100L
IEQUALIFIER_REPEAT	0x0200L
IEQUALIFIER_INTERRUPT	0x0400L
IEQUALIFIER_MULTIBROADCAST	0x0800L
IEQUALIFIER_MIDBUTTON	0x1000L
IEQUALIFIER_RBUTTON	0x2000L
IEQUALIFIER_LEFTBUTTON	0x4000L
IEQUALIFIER_RELATIVEMOUSE	0x8000L

The `input.device` driver allows division in the key groups, so you can recognize the left <Shift>, right <Shift> and <Caps Lock> keys. You can also find out whether information came from the numeric keypad, and if the key found is in repeat mode. You can check the mouse buttons for this data.

If you prefer to work with codes processed according to the keyboard table, then `Class` returns a message of type `VANILLAKEY`. Under `Code` we find the translated character code. This option is the simplest. It doesn't matter which method you use in the program:

```
if (MessageClass & RAWKEY) RAW_Evaluation(Code, Qualifier);

if (MessageClass & VANILLAKEY) VANILLA_Evaluation((char)Code);
```

Various differences appear in reading the code received. While `VANILLAKEY` can simply work with the characters, `RAWKEY` must first consult the keyboard table.

```
VANILLA_Analysis(ZChar)
char ZChar;
{
```

```
    Print(ZChar);
    InputTextBuffer(ZChar);

    ...

}


RAW_Analysis(Code, Qualifier);
USHORT Code, Qualifier;
{
    /* 1. Conversion according to the table */

    ZChar= KeyTab[Qualifier][Code];

    /* or 2. Conversion with individual differences */

    switch(Qualifier)
    {
        case IEQUALIFIER_LSHIFT :
            switch(Code)
                {
                    /* Evaluate left Shift key */
                }
            break;

        case IEQUALIFIER_RSHIFT :
            switch(Code)
                {
                    /* Evaluate right Shift key */
                }
            break;

        switch(Code)
            {
                /* Evaluate without Shift key */
            }
    }
}
```

3.7.2.7 The disk drive

Let's look back at the file select box. This should display the directory of the inserted disk. What happens if you remove the disk from the drive and insert another disk? Nothing! The same view with the same files appears. That's wrong—we should be able to read the new disk's directory.

Two disk flags solve this problem. `DISKREMOVED` tells us when a disk is removed, allowing the file table to be deleted. `DISKINSERTED` starts the read operation that analyzes the new directory.

```
/* 3.7.2.7.a.DISKRMV/INS */
if (MessageClass & DISKREMOVED)
   {
      /* Clear table */
   }

if (MessageClass & DISKINSERTED)
   {
      /* Read new table */
   }
```

3.8 Menus

Has this ever happened to you in any computer language? You write a program which does everything for the user except make coffee in the morning. You've written subroutines to handle every detail, every error, every contingency. The user has more functions in this program than he knows what to do with. But you don't know how to present all these functions in a form that the user can easily learn and remember.

We could use the keyboard exclusively, then draw a keyboard overlay to help the user remember. That's kind of silly, since the Amiga has Intuition. Maybe we can code user access to functions and commands through gadgets. That's better, but since we still need room for an input and output window pair, where do we put the gadgets?

It doesn't matter to the Amiga what you use. Gadgets are good for limited access, but *menus* allow the most user-friendly environment for accessing program commands and functions.

The menus offered by Intuition let us divide all of the functions of our program into groups. Most of these functions go under menu titles named `File` and `Edit`. Each title lists menu items with names applicable to the titles. The user selects a function by pressing and holding the right mouse button, moving the mouse pointer to the menu title, moving the pointer down to the desired item and releasing the right mouse button. The Workbench menus were probably the first menus you ever saw on the Amiga.

We'll write program code in this section that creates multiple menus with multiple items. Some of the menu items even have submenu items. These menus make use of all of the menu options.

3.8.1 General construction of a menu

It is important to know which structure the menus have, so that we can learn our limitations in menu creation. Let's look at the Workbench menus as our first examples.

When the user presses the right mouse button the title bar changes to the menu strip, which displays three names. These are the names of each menu title. The strip itself uses the font used by the current screen. We can program the title text, similar to setting the exact pixel in the X direction.

Now we consider the first menu: The Workbench menu has six functions, none of which can be selected. All of the text appears in ghost print. If you click on an icon then select the Workbench menu, many menu items appear in readable print (you can select one of these).

As the second example we'll use an imaginary drawing program for the menu strip. This menu uses graphic elements—menus can consist of more than lines of text. We can use graphics overall. In addition, we aren't limited to one line of text/graphic menu item data—we can determine how many texts or graphics per line appear per menu item. Be careful not to overlap the individual items!

3.8.2 Making a menu

You now have an idea of what menus can do, thanks to the two examples listed above. Your first goal is to create your own menu strip. We would like to open a new screen with some additional arguments. We need a window, without which a menu strip cannot exist since the two are interrelated.

The following listing accesses a standard program for screens and contains additional font settings:

```
/*************************************
 *                                   *
 * Program : Baseprogram.c           *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb      2/ 3/1988  Empty Menu-Strip*
 *                                   *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Screen        *FirstScreen;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct TextAttr ScreenFont =
   {
   (STRPTR)"topaz.font",
   TOPAZ_SIXTY,
```

```
    FS_NORMAL,
    FPF_ROMFONT
    };

struct NewScreen FirstNewScreen =
    {
    0, 0,                   /* LeftEdge, TopEdge    */
    640, 200,               /* Width, Height        */
    3,                      /* Depth                */
    0, 1,                   /* DetailPen, BlockPen */
    HIRES,                  /* ViewModes            */
    CUSTOMSCREEN,           /* Type                 */
    &ScreenFont,            /* Font                 */
    (UBYTE *)"Screen Test",
    NULL,                   /* Gadgets              */
    NULL,                   /* CustomBitMap         */
    };

struct NewWindow FirstNewWindow =
    {
    0, 0,                   /* LeftEdge, TopEdge    */
    640, 100,               /* Width, Height        */
    0, 1,                   /* DetailPen, BlockPen */
    CLOSEWINDOW,            /* IDCMP Flags          */
    WINDOWDEPTH |           /* Flags                */
    WINDOWSIZING |
    WINDOWDRAG |
    WINDOWCLOSE |
    SMART_REFRESH,
    NULL,                   /* First Gadget         */
    NULL,                   /* CheckMark            */
    (UBYTE *)"Test Menu-Strip",
    NULL,                   /* Screen (kommt noch) */
    NULL,                   /* BitMap               */
    100, 50,                /* Min Width, Height    */
    640, 200,               /* Max Width, Height    */
    CUSTOMSCREEN,           /* Type                 */
    };

main()
    {
    ULONG MessageClass;
    USHORT code;

    struct Message *GetMsg();

    Open_All();

    FOREVER
        {
        if ((message = (struct IntuiMessage *)
             GetMsg(FirstWindow->UserPort)) == NULL)
            {
            Wait(1L << FirstWindow->UserPort->mp_SigBit);
            continue;
            }
```

```
        MessageClass = message->Class;
        code = message->Code;

        ReplyMsg(message);
        switch (MessageClass)
              {
              case CLOSEWINDOW : Close_All();
                                 exit(TRUE);
                                 break;
              }
        }
    }


/*********************************************
 *                                           *
 * Function: Library, Screen and Window open *
 * ========================================= *
 *                                           *
 * Author: Date:       Comments:             *
 * ------  ----------  ----------            *
 * Wgb     10/16/1987  functions!            *
 *                                           *
 *                                           *
 *********************************************/

Open_All()
    {
    void          *OpenLibrary();
    struct Window *OpenWindow();
    struct Screen *OpenScreen();

    if (!(IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L)))
        {
        printf("Intuition Library not found!\n");
        Close_All();
        exit(FALSE);
        }

    if (!(FirstScreen = (struct Screen *)
        OpenScreen(&FirstNewScreen)))
        {
        printf("No screen page!\n");
        Close_All();
        exit(FALSE);
        }

    FirstNewWindow.Screen = FirstScreen;

    if (!(FirstWindow = (struct Window *)
        OpenWindow(&FirstNewWindow)))
        {
        printf("Window will not open!\n");
        Close_All();
        exit(FALSE);
```

```
    }

  }

/*************************************
 *                                   *
 * Function: Close all               *
 * ================================= *
 *                                   *
 * Author: Date:        Comments:    *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  Window, Screen *
 *                     and Intuition *
 *                                   *
 *************************************/

Close_All()
   {
   if (FirstWindow)     CloseWindow(FirstWindow);
   if (FirstScreen)     CloseScreen(FirstScreen);
   if (IntuitionBase)   CloseLibrary(IntuitionBase);
   }
```

3.8.2.1 Filling the menu strip

This test program will help us construct the menu strip. For this we need a structure for each menu. In addition to the menu titles, the menu structure contains a flag, coordinate statements and two left pointers.

```
/* 3.8.2.1.a.menustruct */
struct Menu
{
0x00  00  struct Menu *NextMenu;
0x04  04  SHORT LeftEdge;
0x06  06  SHORT TopEdge;
0x08  08  SHORT Width;
0x0A  10  SHORT Height;
0x0C  12  USHORT Flags;
0x0E  14  BYTE *MenuName;
0x0F  15  struct MenuItem *FirstItem;
0x13  19  SHORT JazzX;
0x15  21  SHORT JazzY;
0x17  23  SHORT BeatX;
0x19  25  SHORT BeatY;
0x1B
};
```

Structure description

`*NextMenu` The menu structure contains a pointer to the result structure, like so many Intuition structures. This pointer helps in the chaining of multiple menus.

`LeftEdge, Width`
These values describe the position and width of the select box. This is the region in which the mouse pointer must lie until the menu becomes active.

`TopEdge, Height`
These values aren't currently implemented. Later versions may allow graphics in the menu strip.

`Flags` This flag determines whether or not the entire menu should be selectable. If not, all of the menu items appear in ghost print. To allow the choice, the `MENUENABLED` flag is set. Intuition sets the `MIDRAWN` flag if the menu is chosen by the user.

`*MenuName` This pointer points to the menu titles. You may only enter a string here, which appears in the default font of the screen.

`*MenuItem` Here we find the chaining together with another structure. It contains all of the information for the individual menu items.

All other variables are used from Intuition and do not need initialization.

We'll create three menus in our test program to start. The first menu, called `Files`, should support all disk functions. It can be used in any program because disk access is important in most programs. The second menu will be called `Style` and the third menu, `Graphics`.

Here are the three structures. All you need to do is type them in and append them to the test program, before the `main()` function.

```
/* 3.8.2.1.b. menu_titles */
struct Menu Graphics =
{
   NULL,
   220, 0,
   90, 10,
   MENUENABLED,
   (BYTE *) "Graphics",
   NULL
};

struct Menu Style =
{
```

```
    &Graphics,
    100, 0,
    70, 10,
    MENUENABLED,
    (BYTE *) "Style",
    NULL
};

struct Menu File =
{
    &Style,
    10, 0,
    60, 10,
    MENUENABLED,
    (BYTE *) "File",
    NULL
};
```

Now insert two new commands in our `_All()` function. First we must place this menu strip in the window. The Intuition function `SetMenuStrip()` performs this task. It has the following format:

```
Success = SetMenuStrip(Window, Menu);
   D0         -264         A0      A1
```

The `Window` lists the window and its address. The menu strip is already active. You can work with it as long as the window is active. Our `Open_All()` function needs the following after the `OpenWindow()` function:

```
SetMenuStrip(FirstWindow, &File);
```

We have not paid attention to the return value `Success` because it always reads true.

The `Close_All()` function needs:

```
if (FirstWindow) ClearMenuStrip(FirstWindow);
```

The general format for `ClearMenuStrip()` is:

```
ClearMenuStrip(window);
     -54          A0
```

Remember that you must remove the menu strip before closing the window, otherwise it could lead to a system crash. In addition, you must remove the menu strip from the window before any menu change. After the change it can again be added with `SetMenuStrip()`.

Program description

After starting the program a screen with a window appears. As long as the window is inactive, you can't reach the menu strip using the right mouse button. When you activate the window, the menu strip also becomes visible. You'll see three menu titles, without items (we'll fix that). Click on the window's close gadget to end the problem.

3.8.2.2 The `MenuItem` structure

Now we'll complete the first menu's design. We'll use text because they're easiest to manipulate. We need an `IntuiText` structure for each text:

```
/* 3.8.2.2.a. intuitext */
struct TextAttr Font =
   {
   (STRPTR)"topaz.font",
   TOPAZ_SIXTY,
   FS_NORMAL,
   FPF_ROMFONT
   };

struct IntuiText LoadText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Load", NULL
   };

struct IntuiText SaveText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Save", NULL
   };

struct IntuiText DeleteText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Delete", NULL
   };

struct IntuiText ProgramendText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Program End", NULL
   };
```

We now have the menu texts but need a structure for each menu item. The `MenuItem` structure reserves all of the important data for each menu item.

```
/* 3.8.2.2.b. menuitem */
struct MenuItem
{
0x00  00  struct MenuItem *NextItem;
0x04  04  SHORT LeftEdge
0x06  06  SHORT TopEdge;
0x08  08  SHORT Width;
0x0A  10  SHORT Height;
0x0C  12  USHORT Flags;
0x0E  14  LONG MutualExclude;
0x12  18  APTR ItemFill;
0x16  22  APTR SelectFill;
0x1A  26  BYTE Command;
```

```
0x1B  27  struct MenuItem *SubItem;
0x1F  31  USHORT NextSelect;
0x21  33
};
```

Structure description

*NextItem — This pointer points to the next menu item. We need it to manage more than one item per menu. All of the menu items are related to this pointer.

LeftEdge, TopEdge — These specify the position of the activation area called the select box. The coordinates are relative to LeftEdge and TopEdge of the menu structure.

Width, Height — These specify the measurements of the select box of the menu items.

MutualExclude — Through the bits of these variables we can see if other menu items should be deactivated by choosing this menu item.

ItemFill — This pointer either points to an Image structure or to an IntuiText structure. You label this with the ITEMTEXT flag (see below for a description of the flags in this structure).

SelectFill — The second pointer contains a pointer to one of the two structures. This occurs only if one of the menu items is activated and the HIGHIMAGE flag is also set (see below for a description of the flags in this structure).

Command — If you define a command under Flags, the ASCII code of the character of the menu item that should be activated is found here (see below for a description of the flags in this structure).

*SubItem — As you may already know, it is possible to insert the menus into another area. This pointer points to the first submenu item of this menu item.

NextSelect — A variable that can be filled after choosing the menu item from Intuition. Then another menu item is chosen and here is its number.

Flags — Here are the flags which specify the appearance of the menu items:

Flags

Normally Intuition assumes a pointer to the `Image` structure under `ItemFill` and `SelectFill`. By setting the flag `ITEMTEXT` you set an `IntuiText` structure.

The `HIGHFLAGS` flag configure the highlighting of the menu item currently under the mouse pointer. When you set `HIGHCOMP`, the select box changes to inverse video. When you set `HIGHBOX`, the select box appears with a surrounding box. `HIGHIMAGE` specifies the display of a new text or graphic. The last flag, `HIGHNONE`, tells Intuition to not do anything.

The `ITEMENABLED` flag indicates whether a specific menu item should be selectable or not. Set `ITEMENABLED` if you want the menu item selectable.

Command keys (keyboard abbreviations) can be assigned to any menu item. Set the `COMMSEQ` flag to assign the item to a key combination.

Intuition also allows the selection and unselection of certain menu items, using the `CHECKIT` flag. The menu item then stands in either an active or inactive state.

Should a selectable/unselectable menu item be unselected when the menu initializes, then you must set the `CHECKED` flag.

Intuition has reserved two additional flags in the variable `Flags`. Specifying a submenu item of a menu item sets the `ISDRAWN` flag. The `HIGHITEM` flag is set when the mouse pointer activates a menu item.

Now, let's start programming our first menu items. You have hopefully read the `IntuiText` structure data above and typed these structures in. We have four `MenuItem` structures to add to our program:

```
/* 3.8.2.2.c.menuitems */
struct MenuItem Programend =
{
   NULL,
   1, 40,
   120, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&ProgramendText,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Delete =
{
   &Programend,
```

```
    1, 26,
    120, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP,
    0,
    (APTR)&DeleteText,
    NULL,
    0,
    NULL,
    0
};

struct MenuItem Save =
{
    &Delete,
    1, 14,
    120, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP,
    0,
    (APTR)&SaveText,
    NULL,
    0,
    NULL,
    0
};

struct MenuItem Load =
{
    &Save,
    1, 2,
    120, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP,
    0,
    (APTR)&LoadText,
    NULL,
    0,
    NULL,
    0
};
```

Here is the complete program for our first menu:

```
/*************************************
 *                                   *
 * Program: Demonstration Menu-Strip *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb      2/ 3/1988  Menu-Strip    *
 *                     one menu built*
 *                                   *
 *************************************/


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

```
struct IntuitionBase *IntuitionBase;
struct Screen        *FirstScreen;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct TextAttr Font =
   {
   (STRPTR)"topaz.font",
   TOPAZ_SIXTY,
   FS_NORMAL,
   FPF_ROMFONT
   };

struct NewScreen FirstNewScreen =
   {
   0, 0,                  /* LeftEdge, TopEdge   */
   640, 200,              /* Width, Height       */
   3,                     /* Depth               */
   0, 1,                  /* DetailPen, BlockPen */
   HIRES,                 /* ViewModes           */
   CUSTOMSCREEN,          /* Type                */
   &Font,                 /* Font                */
   (UBYTE *)"Screen Test",
   NULL,                  /* Gadgets             */
   NULL,                  /* CustomBitMap        */
   };

struct NewWindow FirstNewWindow =
   {
   0, 0,                  /* LeftEdge, TopEdge   */
   640, 100,              /* Width, Height       */
   0, 1,                  /* DetailPen, BlockPen */
   CLOSEWINDOW,           /* IDCMP Flags         */
   WINDOWDEPTH |          /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   SMART_REFRESH,
   NULL,                  /* First Gadget        */
   NULL,                  /* CheckMark           */
   (UBYTE *)"Test Menu-Strip",
   NULL,                  /* Screen (kommt noch) */
   NULL,                  /* BitMap              */
   100, 50,               /* Min Width, Height   */
   640, 200,              /* Max Width, Height   */
   CUSTOMSCREEN,          /* Type                */
   };

struct IntuiText LoadText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Load", NULL
   };
```

```
struct IntuiText SaveText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Save", NULL
   };

struct IntuiText DeleteText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Delete", NULL
   };

struct IntuiText ProgramendText =
   {
   4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Program end", NULL
   };

struct MenuItem Programend =
{
   NULL,
   1, 40,
   120, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&ProgramendText,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Delete =
{
   &Programend,
   1, 26,
   120, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&DeleteText,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Save =
{
   &Delete,
   1, 14,
   120, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&SaveText,
   NULL,
   0,
   NULL,
   0
};
```

```
struct MenuItem Load =
{
   &Save,
   1, 2,
   120, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&LoadText,
   NULL,
   0,
   NULL,
   0
};


struct Menu Graphic =
{
   NULL,
   220, 0,
   90, 10,
   MENUENABLED,
   (BYTE *) "Graphics",
   NULL
};

struct Menu Style =
{
   &Graphic,
   100, 0,
   70, 10,
   MENUENABLED,
   (BYTE *) "Style",
   NULL
};

struct Menu File =
{
   &Style,
   10, 0,
   60, 10,
   MENUENABLED,
   (BYTE *) "File",
   &Load
};



main()
   {
   ULONG MessageClass;
   USHORT code;

   struct Message *GetMsg();

   Open_All();
```

```
   FOREVER
      {
      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
         Wait(1L << FirstWindow->UserPort->mp_SigBit);
         continue;
         }
      MessageClass = message->Class;
      code = message->Code;

      ReplyMsg(message);
      switch (MessageClass)
            {
            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;
            }
      }
   }


/*********************************************
 *                                           *
 * Function: Library, Screen and Window open *
 * ========================================= *
 *                                           *
 * Author: Date:       Comments:             *
 * ------  ----------  ----------            *
 * Wgb     16.10.1987  functioning!          *
 *                                           *
 *                                           *
 *********************************************/

Open_All()
   {
   void          *OpenLibrary();
   struct Window *OpenWindow();
   struct Screen *OpenScreen();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstScreen = (struct Screen *)
       OpenScreen(&FirstNewScreen)))
      {
      printf("No screen page!\n");
      Close_All();
      exit(FALSE);
      }
```

```
    FirstNewWindow.Screen = FirstScreen;

    if (!(FirstWindow = (struct Window *)
        OpenWindow(&FirstNewWindow)))
       {
       printf("Window will not open!\n");
       Close_All();
       exit(FALSE);
       }

    SetMenuStrip(FirstWindow, &File);

    }


/*************************************
 *                                   *
 * Function: Close all               *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  Window, Screen *
 *                     and Intuition *
 *                                   *
 *************************************/

Close_All()
    {
    if (FirstWindow)      ClearMenuStrip(FirstWindow);
    if (FirstWindow)      CloseWindow(FirstWindow);
    if (FirstScreen)      CloseScreen(FirstScreen);
    if (IntuitionBase)    CloseLibrary(IntuitionBase);
    }
```

3.8.2.3 Adding extras to menu design

Most people would be satisfied with the above structures, definitions, texts and settings. Not us. We currently have two itemless menus, and one menu without any submenu items. Pretty boring stuff. It needs a little showmanship, as well as some user conveniences. Let's go through the data step by step, and make full use of Intuition.

First is convenience (which is actually bottom priority in this instance). Imagine yourself in the following situation: You are the long time user of a program and know all its possibilities, but must always take the mouse in your hand to go through the menus to select one item.

This mouse-to-keyboard-to-mouse switching is a pain. It would be much easier if you could call the menu item from the keyboard as well as from the menu. You can do it: Just set the COMMSEQ flag in the Flags array of the MenuItem structure, and to give the Command argument the ASCII value of the key. The <Amiga> key and our character appear to the right of the menu item. Widen the menu item box by at least four characters (40 pixels): One character for space, two characters for the A character, and one character for the command key itself. The command key is always called by pressing the right <Amiga> key, then the key.

Widen all of the item boxes by 40 pixels and set the COMMSEQ flags in the Flag arrays in each structure (don't forget to include a letter for each command key). We recommend the following: "L" for Load, "S" for Save, "D" for Delete, and "E" for end program. Now look at the MenuItem structures:

```
/* 3.8.2.3.a.menuitem*/
struct MenuItem Programend =
{
   NULL,
   1, 40,
   170, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&ProgramendText,
   NULL,
   0x45,
   NULL,
   0
};

struct MenuItem Delete =
{
   &Programend,
   1, 26,
   170, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&DeleteText,
   NULL,
   0x44,
   NULL,
   0
};

struct MenuItem Save =
{
   &Delete,
   1, 14,
   170, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&SaveText,
```

```
    NULL,
    0x53,
    NULL,
    0
};

struct MenuItem Load =
{
    &Save,
    1, 2,
    170, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&LoadText,
    NULL,
    0x4C,
    NULL,
    0
};
```

Note: Remember that Intuition differentiates between upper case and lower case letters. For example, you can use right <Amiga><S> for one command key, and <Shift> right <Amiga><S> for a different command key.

Highlighting The menu items alone aren't enough. A menu item is displayed in inverse video to indicate it's selected, but we can add still more "bells 'n' whistles." For example, we can program the menu to change the menu item text when an item is selected. Take the Delete item: When the item changes to inverse video, we can make the text change to something like "REALLY?" or "Are you sure?", to remind the user of what he's selecting. The `HIGHFLAGS` make this text swap possible.

For this we need a new `IntuiText` structure.We'll also change the color at the same time:

```
/* 3.8.2.3.b.swaptext*/
struct IntuiText DeleteText =
    {
    5, 1, JAM2, 1, 1, &Font, (UBYTE *)"Delete ", NULL
    };

struct IntuiText DeleteTextII =
    {
    5, 1, JAM2, 1, 1, &Font, (UBYTE *)"REALLY?", NULL
    };
```

You can change the text position or the font. Many options are possible with this code.

We must remember to change the `MenuItem` structure. For one, the `HIGHIMAGE` flag must replace the `HIGHCOMP` flag, and then the `SelectFill` pointer must be supplied with the pointer to our text:

```
/*3.8.2.3.c.highimage_struct*/
struct MenuItem Delete =
{
   &Programend,
   1, 26,
   170, 10,
   ITEMTEXT | ITEMENABLED | HIGHIMAGE | COMMSEQ,
   0,
   (APTR)&DeleteText,
   (APTR)&DeleteTextII,
   0x44,
   NULL,
   0
};
```

We kept the letter command sequence because it is always practical. Compile, link and test the menu choice once.

For graphic menu display, we suggest you look to the other two HIGHFLAG modes for help (HIGHNONE and HIGHBOX). The HIGHBOX flag draws a box around the selected menu item. Perhaps you want some menu items to not have boxes drawn around them. Use the HIGHNONE flag. For example, if there are many functions in a menu, it would be good if these were also arranged so that some can be accessed in certain cases. If you don't want this, you can simply mark a graphic or text menu item with a line and not have the program read it.

HIGHNONE actually helps you avoid any program abuse from the user. For example, why enable the Save item at the start of an editor program, if there's nothing in memory to save yet? We can enable or disable the Save item as needed using the ITEMENABLE flag. Here's the "new and improved" Save structure:

```
/*3.8.2.3.d.newsave_struct*/
struct MenuItem Save =
{
   &Delete,
   1, 14,
   170, 10,
   ITEMTEXT | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&SaveText,
   NULL,
   0x53,
   NULL,
   0
};
```

3.8.2.4 Submenus

We're not done yet—we need to go deeper yet in the menu strip. The submenus are next. These menus appear when we select a menu item which requires additional parameters or information.

How do we access a submenu? The `MenuItem` structure performs this task (see Section 3.8.2.2). We find a pointer to a `SubItem` there—that's our entrance to the submenu. If you insert the pointer to a new `MenuItem` structure, that acts as the beginning of the new submenu.

These methods give you the option of expanding the `File` menu. With a `File` menu in a word processor, you may want to enter the file type of the text you want loaded/saved. Enter the submenu. For our example you can choose from three filetypes: Text (resident text format used by the word processor), ASCII (unformatted text file to allow transfer to any computer), and IFF (Interchange File Format, Electronic Arts universal file exchange format).

```
/*3.8.2.4.a.text_stuff*/
struct IntuiText TextText =
   {
   3, 1, JAM2, 1, 1, &Font, (UBYTE *)"Text", NULL
   };

struct IntuiText ASCIIText =
   {
   3, 1, JAM2, 1, 1, &Font, (UBYTE *)"ASCII", NULL
   };

struct IntuiText IFFText =
   {
   3, 1, JAM2, 1, 1, &Font, (UBYTE *)"IFF", NULL
   };

struct MenuItem IFF =
{
   NULL,
   140, 22,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&IFFText,
   NULL,
   0x49,
   NULL,
   0
};

struct MenuItem ASCII =
{
```

```
    &IFF,
    140, 12,
    100, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&ASCIIText,
    NULL,
    0x41,
    NULL,
    0
};

struct MenuItem Text =
{
    &ASCII,
    140, 2,
    100, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&TextText,
    NULL,
    0x54,
    NULL,
    0
};

struct MenuItem Load =
{
    &Save,
    1, 2,
    170, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP,
    0,
    (APTR)&LoadText,
    NULL,
    0,
    &Text,
    0
};
```

The same submenu must be expanded with the `Save` function. You must expand the `MenuItem` structure with the pointer to the submenus and remove the old command sequence:

```
/*3.8.2.4.b.bigsave_struct*/
struct MenuItem Save =
{
    &Delete,
    1, 14,
    170, 10,
    ITEMTEXT | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&SaveText,
    NULL,
    0,
```

```
    &SaverText,
    0
};
```

Unfortunately, the `SubItems` must also be redefined because the old command sequences cannot be used twice:

```
/*3.8.2.4.c.IFFsave_struct*/
struct MenuItem SaveIFF =
{
    NULL,
    140, 22,
    100, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&IFFText,
    NULL,
    0x46,
    NULL,
    0
};

struct MenuItem SaveASCII =
{
    &SaveIFF,
    140, 12,
    100, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&ASCIIText,
    NULL,
    0x43,
    NULL,
    0
};

struct MenuItem SaverText =
{
    &SaveASCII,
    140, 2,
    100, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&TextText,
    NULL,
    0x51,
    NULL,
    0
};
```

We can use the `IntuiText` structures a second time without thinking about it.

3.8.2.5 Style menu tutorial

You now have the essentials for programming your own menus. All we need to do now is the menu descriptions. You can then define the structures as you wish, and link them together.

The menu should consist of six text types. Each item should be accessible through a command key. The following texts types are standard: `Normal`, `Italics`, `Bold`, `Inverse`, `80 characters` and `60 characters`. Each menu item must be defined as an `IntuiText` structure with its own `TextAttr` structure. In the `TextAttr` structure you give the respective text types. Now the user has a simple choice and can immediately select a text style.

When the mouse is activate, the user should be able to see which text types are currently enabled. The checkmark will serve our purposes. Set the `CHECKIT` flag in the `Flags` variable. The checkmark character must precede the text in the `IntuiText` structure (two spaces preceding the text should be enough). We also recommend that the `MENUTOGGLE` flag be used to enable and disable text styles.

Next we must turn toward `MutualExclude`. This flag lets us change checkmarked menu items to unchecked items, and unchecked items to checkmarked items. Confused? It's actually very simple and very useful. Look at a concrete example: if the user selects the `Normal` text type, this disables all other text types. We accomplish this by using `MutualExclude`. The same goes for the two text sizes. Only one can be used at a time; the other must be deactivated.

`MutualExclude` works as follows: We have a long value for which each bit represents a menu item. All of the menu items marked with a set bit should be deactivated when a particular menu item is selected. Bit 0 corresponds to the first menu item, bit 1 to the second menu item, etc.

3.8.2.6 Graphic menus

After adding type style choices to the `Style` menu, you have the option of adding graphics and color choices to the `Graphic` menu. This menu goes well in a drawing or paint program.

The menu consists of two menu items. The first, `Pen`, has a submenu with two more subitems: A fine point brush graphic and a thick point brush graphic. The second menu item, `Palette`, has eight submenu

items which specify the drawing color. This can be useful for both the drawing program and the word processor alike.

Here are the `MenuItem` structures for the `Pen` menu:

```
/*3.8.2.6.a.brushmenu_struct*/
struct MenuItem PenII =
{
   NULL,
   90, 2,
   10, 10,
   ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&PenIIImage,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem PenI =
{
   &PenII,
   102, 2,
   10, 10,
   ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&PenIImage,
   NULL,
   0,
   NULL,
   0
};
```

And the `MenuItem` structure for the `Graphic` menu:

```
/*3.8.2.6.b.brushmenuitem */
struct IntuiText PenText =
   {
   0, 1, JAM2, 1, 1, &Font, (UBYTE *)"Brush", NULL
   };

struct MenuItem Pen =
{
   &Colour,
   2, 2,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&PenText,
   NULL,
   0,
   &PenI,
   0
};
```

Here are the two Image structures and their values:

```
/*3.8.2.6.c.imagedata*/
USHORT PenIData[] =
   {
   0x00, 0x00,
   0x01, 0xF0,
   0x00, 0x00,
   0x00, 0x00,
   };

USHORT PenIIData[] =
   {
   0x00, 0x00,
   0x01, 0xF0,
   0x01, 0xF0,
   0x00, 0x00,
   };

struct Image PenIImage =
   {
   1, 1,
   8, 8,
   1,
   &PenIData[0],
   2, 0,
   NULL
   };

struct Image PenIIImage =
   {
   1, 1,
   8, 8,
   2,
   &PenIIData[0],
   2, 0,
   NULL
   };
```

The data for the color choice submenu follow. We define a MenuItem structure for each menu item. The submenu represents every color through IntuiText structure. First, the main menu:

```
/*3.8.2.6.d.colortext*/
struct IntuiText ColornText =
   {
   0, 1, JAM2, 1, 1, &Font, (UBYTE *)"Colors", NULL
   };

struct MenuItem Colorn =
{
   NULL,
   2, 14,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
```

```
    0,
    (APTR)&ColornText,
    NULL,
    0,
    &Color0,
    0
};
```

For the colors we take `IntuiText` structures which contain blank spaces only. The background color changes so that color bars appear instead of spaces. We show only the first three structures; the others are easy to construct from there:

```
/*3.8.2.6.e.col0text*/
struct IntuiText Col0Text =
    {
    0, 0, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };

struct IntuiText Col1Text =
    {
    0, 1, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };

struct IntuiText Col2Text =
    {
    0, 2, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };
...
```

Define the `MenuItem` submenu structure based on the same principles:

```
...

/*3.8.2.6.f.color2*/
struct MenuItem Color2 =
{
    &Color3,
    90, 26,
    50, 10,
    ITEMTEXT | ITEMENABLED | HIGHBOX,
    0,
    (APTR)&Col2Text,
    NULL,
    0,
    NULL,
    0
};

struct MenuItem Color1 =
{
    &Color2,
    90, 14,
    50, 10,
    ITEMTEXT | ITEMENABLED | HIGHBOX,
```

```
    0,
    (APTR)&Col1Text,
    NULL,
    0,
    NULL,
    0
};

struct MenuItem Color0 =
{
    &Color1,
    90, 2,
    50, 10,
    ITEMTEXT | ITEMENABLED | HIGHBOX,
    0,
    (APTR)&Col0Text,
    NULL,
    0,
    NULL,
    0
};
```

Here is an example program:

```
/*************************************
 *                                   *
 * Program: Demonstration Menu-Strip *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb      2/ 3/1988  Menu-Strip    *
 *                                   *
 *                                   *
 *************************************/


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


struct IntuitionBase *IntuitionBase;
struct Screen        *FirstScreen;
struct Window        *FirstWindow;
struct IntuiMessage  *message;


struct TextAttr Font =
    {
    (STRPTR)"topaz.font",
    TOPAZ_SIXTY,
    FS_NORMAL,
    FPF_ROMFONT
    };

struct NewScreen FirstNewScreen =
```

```
    {
    0, 0,                   /* LeftEdge, TopEdge    */
    640, 200,               /* Width, Height        */
    3,                      /* Depth                */
    0, 1,                   /* DetailPen, BlockPen  */
    HIRES,                  /* ViewModes            */
    CUSTOMSCREEN,           /* Type                 */
    &Font,                  /* Font                 */
    (UBYTE *)"Screen Test",
    NULL,                   /* Gadgets              */
    NULL,                   /* CustomBitMap         */
    };

struct NewWindow FirstNewWindow =
    {
    0, 0,                   /* LeftEdge, TopEdge    */
    640, 100,               /* Width, Height        */
    0, 1,                   /* DetailPen, BlockPen  */
    CLOSEWINDOW,            /* IDCMP Flags          */
    WINDOWDEPTH |           /* Flags                */
    WINDOWSIZING |
    WINDOWDRAG |
    WINDOWCLOSE |
    SMART_REFRESH,
    NULL,                   /* First Gadget         */
    NULL,                   /* CheckMark            */
    (UBYTE *)"Test Menu-Strip",
    NULL,                   /* Screen (kommt noch)  */
    NULL,                   /* BitMap               */
    100, 50,                /* Min Width, Height    */
    640, 200,               /* Max Width, Height    */
    CUSTOMSCREEN,           /* Type                 */
    };


USHORT PenIData[] =
    {
    0x00, 0x00,
    0x01, 0xF0,
    0x00, 0x00,
    0x00, 0x00,
    };

USHORT PenIIData[] =
    {
    0x00, 0x00,
    0x01, 0xF0,
    0x01, 0xF0,
    0x00, 0x00,
    };

struct Image PenIImage =
    {
    1, 1,
    8, 8,
    1,
```

```
    &PenIData[0],
    2, 0,
    NULL
    };

struct Image PenIIImage =
    {
    1, 1,
    8, 8,
    2,
    &PenIIData[0],
    2, 0,
    NULL
    };


struct IntuiText ColorText =
    {
    0, 1, JAM2, 1, 1, &Font, (UBYTE *)"Color", NULL
    };


struct IntuiText PenText =
    {
    0, 1, JAM2, 1, 1, &Font, (UBYTE *)"Pen", NULL
    };

struct IntuiText LoadText =
    {
    4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Load", NULL
    };

struct IntuiText SaveText =
    {
    4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Save", NULL
    };

struct IntuiText DeleteText =
    {
    5, 1, JAM2, 1, 1, &Font, (UBYTE *)"Delete ", NULL
    };

struct IntuiText DeleteTextII =
    {
    5, 1, JAM2, 1, 1, &Font, (UBYTE *)"REALLY?", NULL
    };


struct IntuiText ProgramendText =
    {
    4, 1, JAM2, 1, 1, &Font, (UBYTE *)"Program end", NULL
    };

struct IntuiText TextText =
    {
    3, 1, JAM2, 1, 1, &Font, (UBYTE *)"Text", NULL
```

```
    };

struct IntuiText ASCIIText =
    {
    3, 1, JAM2, 1, 1, &Font, (UBYTE *)"ASCII", NULL
    };

struct IntuiText IFFText =
    {
    3, 1, JAM2, 1, 1, &Font, (UBYTE *)"IFF", NULL
    };

struct IntuiText Col0Text =
    {
    0, 0, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };

struct IntuiText Col1Text =
    {
    0, 1, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };

struct IntuiText Col2Text =
    {
    0, 2, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };

struct IntuiText Col3Text =
    {
    0, 3, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };

struct IntuiText Col4Text =
    {
    0, 4, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };

struct IntuiText Col5Text =
    {
    0, 5, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };
struct IntuiText Col6Text =
    {
    0, 6, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };

struct IntuiText Col7Text =
    {
    0, 7, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };

struct IntuiText Col8Text =
    {
    0, 8, JAM2, 1, 1, &Font, (UBYTE *)"     ", NULL
    };
```

```
struct MenuItem PenII =
{
   NULL,
   90, 2,
   10, 10,
   ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&PenIIImage,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem PenI =
{
   &PenII,
   102, 2,
   10, 10,
   ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&PenIImage,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Color8 =
{
   NULL,
   90, 98,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col8Text,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Color7 =
{
   &Color8,
   90, 86,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col7Text,
   NULL,
   0,
   NULL,
   0
};
```

```
struct MenuItem Color6 =
{
   &Color7,
   90, 74,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col6Text,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Color5 =
{
   &Color6,
   90, 62,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col5Text,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Color4 =
{
   &Color5,
   90, 50,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col4Text,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Color3 =
{
   &Color4,
   90, 38,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col3Text,
   NULL,
   0,
   NULL,
   0
};
```

```
struct MenuItem Color2 =
{
   &Color3,
   90, 26,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col2Text,
   NULL,
   0,
   NULL,
   0
};


struct MenuItem Color1 =
{
   &Color2,
   90, 14,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col1Text,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Color0 =
{
   &Color1,
   90, 2,
   50, 10,
   ITEMTEXT | ITEMENABLED | HIGHBOX,
   0,
   (APTR)&Col0Text,
   NULL,
   0,
   NULL,
   0
};

struct MenuItem Color =
{
   NULL,
   2, 14,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&ColorText,
   NULL,
   0,
   &Color0,
   0
};
```

```
struct MenuItem Pen =
{
   &Color,
   2, 2,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP,
   0,
   (APTR)&PenText,
   NULL,
   0,
   &PenI,
   0
};


struct MenuItem IFF =
{
   NULL,
   140, 22,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&IFFText,
   NULL,
   0x49,
   NULL,
   0
};

struct MenuItem ASCII =
{
   &IFF,
   140, 12,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&ASCIIText,
   NULL,
   0x41,
   NULL,
   0
};

struct MenuItem Text =
{
   &ASCII,
   140, 2,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&TextText,
   NULL,
   0x54,
   NULL,
   0
```

```
};


struct MenuItem SaveIFF =
{
   NULL,
   140, 22,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&IFFText,
   NULL,
   0x46,
   NULL,
   0
};

struct MenuItem SaveASCII =
{
   &SaveIFF,
   140, 12,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&ASCIIText,
   NULL,
   0x43,
   NULL,
   0
};

struct MenuItem SaverText =
{
   &SaveASCII,
   140, 2,
   100, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&TextText,
   NULL,
   0x51,
   NULL,
   0
};

struct MenuItem Programend =
{
   NULL,
   1, 40,
   170, 10,
   ITEMTEXT | ITEMENABLED | HIGHCOMP | COMMSEQ,
   0,
   (APTR)&ProgramendText,
   NULL,
   0x45,
   NULL,
```

```
    0
};

struct MenuItem Delete =
{
    &Programend,
    1, 26,
    170, 10,
    ITEMTEXT | ITEMENABLED | HIGHIMAGE | COMMSEQ,
    0,
    (APTR)&DeleteText,
    (APTR)&DeleteTextII,
    0x44,
    NULL,
    0
};


struct MenuItem Save =
{
    &Delete,
    1, 14,
    170, 10,
    ITEMTEXT | HIGHCOMP | COMMSEQ,
    0,
    (APTR)&SaveText,
    NULL,
    0x53,
    &SaverText,
    0
};

struct MenuItem Load =
{
    &Save,
    1, 2,
    170, 10,
    ITEMTEXT | ITEMENABLED | HIGHCOMP,
    0,
    (APTR)&LoadText,
    NULL,
    0,
    &Text,
    0
};

struct Menu Graphic =
{
    NULL,
    220, 0,
    90, 10,
    MENUENABLED,
    (BYTE *) "Graphics",
    &Pen
};
```

```
struct Menu Design =
{
   &Graphic,
   100, 0,
   70, 10,
   MENUENABLED,
   (BYTE *) "Design",
   NULL
};

struct Menu File =
{
   &Design,
   10, 0,
   60, 10,
   MENUENABLED,
   (BYTE *) "File",
   &Load
};



main()
   {
   ULONG MessageClass;
   USHORT code;

   struct Message *GetMsg();

   Open_All();

   FOREVER
      {
      if ((message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort)) == NULL)
         {
         Wait(1L << FirstWindow->UserPort->mp_SigBit);
         continue;
         }
      MessageClass = message->Class;
      code = message->Code;

      ReplyMsg(message);
      switch (MessageClass)
            {
            case CLOSEWINDOW : Close_All();
                               exit(TRUE);
                               break;
            }
      }
   }


/*********************************************
 *                                           *
 * Function: Library, Screen and Window open   *
```

```
 * ========================================== *
 *                                            *
 * Author: Date:       Comments:              *
 * ------  ----------  ----------             *
 * Wgb     16.10.1987  functioning!           *
 *                                            *
 *                                            *
 **********************************************/

Open_All()
   {
   void          *OpenLibrary();
   struct Window *OpenWindow();
   struct Screen *OpenScreen();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstScreen = (struct Screen *)
       OpenScreen(&FirstNewScreen)))
      {
      printf("No screen page!\n");
      Close_All();
      exit(FALSE);
      }

   FirstNewWindow.Screen = FirstScreen;

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&FirstNewWindow)))
      {
      printf("Window will not open!\n");
      Close_All();
      exit(FALSE);
      }

   SetMenuStrip(FirstWindow, &File);

   }


/*************************************
 *                                   *
 * Function: Close all               *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  Window, Screen *
 *                     and Intuition *
 *                                   *
```

```
*************************************/

Close_All()
   {
   if (FirstWindow)      ClearMenuStrip(FirstWindow);
   if (FirstWindow)      CloseWindow(FirstWindow);
   if (FirstScreen)      CloseScreen(FirstScreen);
   if (IntuitionBase)    CloseLibrary(IntuitionBase);
   }
```

3.8.3 Reading the menu strip

The menu strip mentioned in this section represents a universal menu type because it consists of more than one menu, one menu item and more than one submenu item. In addition, we find menu items that release actions applicable to certain items. For this reason the menu strip is suitable for a general test. You can use the following functions in each of your programs, because menu reading is especially flexible.

As for every IDCM access, we need the proper parameters. For this we set the MENUPICK flag in the Window structure IDCMP variable.

Next we test for a menu message in the loop:

```
/*3.8.3.a.menureadforever*/
FOREVER
   {
   if ((message = (struct IntuiMessage *)
       GetMsg(FirstWindow->UserPort)) == NULL)
      {
      Wait(1L << FirstWindow->UserPort->mp_SigBit);
      continue;
      }
   MessageClass = message->Class;
   Code = message->Code;

   ReplyMsg(message);
   switch (MessageClass)
         {
         case CLOSEWINDOW : Close_All();
                            exit(TRUE);
                            break;

         case MENUPICK    : Menu_Analysis(Code);
                            break;
         }
      }
   }
```

When we get a message or type MENUPICK, we branch to the Menu_Analysis function, where we can examine one point after another:

```
/*3.8.3.b.menu analysis*/
Menu_Analysis(Menunumber)
USHORT Menunumber;
{
   USHORT Menu, MenuItem, SubItem, NextMenu;
   struct MenuItem *MenuPoint;

   Menu     = MENUNUM(Menunumber);
   MenuItem = ITEMNUM(Menunumber);
   SubItem  = SUBNUM(Menunumber);

   MenuPoint = ItemAddress(&File, Menunumber);

   switch(Menu)
   {
      case 0 : /* File - Menu */

         break;

      case 1 : /* Style - Menu */

         break;

      case 2 : /* Graphic - Menu */

         break;
   }

   NextMenu = MenuPoint->NextSelect;
   if (NextMenu) Menu_Analysis(NextMenu);

}
```

Here we choose the respective menus. The program calculates the address of the MenuItem structure with the function ItemAddress(). We can read more data out. This uses the new function if the value NextSelect is read from a new menu number. This routine also supports the choosing of multiple menu items.

Now we will go through the reading process, starting with the File menu. Let's lay out four menu items:

```
/*3.8.3.c.menuitems*/
switch(MenuItem)
   {
   case 0 : /* Load */

      break;

   case 1 : /* Save */
```

```
      break;

   case 2 : /* Delete */
      Delete();
      break;

   case 3 : /* Program end */
      Ende = FALSE;
      break;
   }
```

The last two items finish a process. Only the first two must be read for now:

```
/*3.8.3.d.load_save*/
case 0 : /* Load */

   switch(SubItem)
      {
      case 0 : /* Text */
         Load(TEXT);
         break;
      case 1 : /* ASCII */
         Load(ASCII);
         break;
      case 2 : /* IFF */
         Load(IFF);
         break;
      }
   break;

case 1 : /* Save */

   switch(SubItem)
      {
      case 0 : /* Text */
         Save(TEXT);
         break;
      case 1 : /* ASCII */
         Save(ASCII);
         break;
      case 2 : /* IFF */
         Save(IFF);
         break;
      }
   break;
```

All submenu accesses call the same function. They pass parameters (in this case, self-defined flags). `TEXT` tells the `Load()` or `Save()` structures that it is waiting for the command sequences needed to write out/load normal text. `ASCII` would strip out any formatting and `IFF` would save/load data in Interchange File Format.

The second menu requires an alternate method of reading. We have no submenus, only a main menu where many features interact with one

another. Only the text width is incompatible. Assuming that the program uses one variable for the text type and one for character width, the following function makes sense:

```
/*3.8.3.e.textstyle*/
switch(MenuItem)
   {
   case 0 : /* Normal */
      TextStyle = NULL;
      break;

   case 1 : /* Italics */
      TextStyle ^= ITALIC;
      break;

   case 2 : /* Bold */
      TextStyle ^= BOLD
      break;

   case 3 : /* Inverse */
      TextStyle ^= INVERS;
      break;

   case 4 : /* 80 Characters */
      TextFont = 80L;
      break;

   case 5 : /* 60 Characters */
      TextFont = 60L;
      break;
   }
```

The third menu lets us return to simpler reading. The `Pen` item is first; the `PenNr` variable contains the number of the selected pen. The variable `ColorNr` contains the color number after selecting that color:

```
/*3.8.3.f.brush_color*/
switch(MenuItem)
   {
   case 0 : /* Brush */
      PenNr = SubItem + 1;
      break;

   case 1 : /* Color */
      ColNr = SubItem;
      break;
   }
```

3.8.4 Working with source code utilities

What do you know about source code utilities? These are what we call programs that help the programmer in developing program source code. Many source codes execute complex and large tasks—tasks that are theoretially hard to process because they lean heavily on graphics. The data entry itself isn't difficult, but involve lots of writing and development time. A source code utility decreases the amount of writing time. It provides the necessary definitions in a clear, easy-to-understand format, which the developer may change to suit his/her own needs.

You'll find such utilities very helpful when programming Intuition. You've seen how many parameters are necessary just for a simple window. This grows in complexity with every gadget (by one, two and even three structures). You remember how much typing you had to do to program in the requesters.

It's gotten to the point where writing such large structures yourself stops being feasible. Since a program needs menus only when you have multiple functions available, we need at least two structures per menu. Some programs may feature over 50 menu items. For example, ***BeckerText*** from Abacus has five menus with 42 menu items and 65 submenu items. For this you would have to define 84 `MenuItem` structures and 130 `IntuiText` structures. We can decrease this work using a source code utility program.

PowerWindows® is one such program that relieves a programmer of a lot of work. It allows the programmer to easily create windows, menu strips and gadgets—the three most important elements of Intuition. It will write a commented source code to disk for the programmer. If you are going to be using C on the Amiga a lot, PowerWindow® is a good investment.

Here is an example source text generated by PowerWindow®:

```
struct IntuiText IText1 = {
   3,7,JAM2,          /* front and back text pens and drawmode */
   20,1,           /* XY origin relative to container TopLeft */
   NULL,              /* font pointer or NULL for defaults */
   "      ",                                 /* pointer to text */
   NULL                              /* next IntuiText structure */
};

struct MenuItem SubItem8 = {
   NULL,                              /* next SubItem structure */
   34,62,                          /* XY of Item hitbox relative to
                                        TopLeft of parent hitbox */
   68,10,                          /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,          /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText1,  /* Item render (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
```

```
   NULL,                              /* alternate command-key */
   NULL,                          /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText2 = {
   3,6,JAM2,          /* front and back text pens and drawmode */
   20,1,            /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "      ",                                 /* pointer to text */
   NULL                             /* next IntuiText structure */
};

struct MenuItem SubItem7 = {
   &SubItem8,                         /* next SubItem structure */
   34,52,                            /* XY of Item hitbox relative
                                     to TopLeft of parent hitbox */
   68,10,                            /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,          /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText2,  /* Item render (IntuiText or Image or NULL) */
   NULL,                                      /* Select render */
   NULL,                               /* alternate command-key */
   NULL,                          /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText3 = {
   3,5,JAM2,          /* front and back text pens and drawmode */
   20,1,            /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "      ",                                 /* pointer to text */
   NULL                             /* next IntuiText structure */
};

struct MenuItem SubItem6 = {
   &SubItem7,                         /* next SubItem structure */
   34,42,                            /* XY of Item hitbox relative
                                     to TopLeft of parent hitbox */
   68,10,                            /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,          /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText3,  /* Item render (IntuiText or Image or NULL) */
   NULL,                                      /* Select render */
   NULL,                               /* alternate command-key */
   NULL,                          /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText4 = {
   3,4,JAM2,          /* front and back text pens and drawmode */
   20,1,            /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "      ",                                 /* pointer to text */
   NULL                             /* next IntuiText structure */
};

struct MenuItem SubItem5 = {
   &SubItem6,                         /* next SubItem structure */
   34,32,                            /* XY of Item hitbox relative
                                     to TopLeft of parent hitbox */
   68,10,                            /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,          /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText4, /* Item render  (IntuiText or Image or NULL) */
   NULL,                                      /* Select render */
   NULL,                               /* alternate command-key */
   NULL,                          /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};
```

```
struct IntuiText IText5 = {
   3,3,JAM2,          /* front and back text pens and drawmode */
   20,1,           /* XY origin relative to container TopLeft */
   NULL,                /* font pointer or NULL for defaults */
   "      ",                                 /* pointer to text */
   NULL                             /* next IntuiText structure */
};

struct MenuItem SubItem4 = {
   &SubItem5,                          /* next SubItem structure */
   34,22,                             /* XY of Item hitbox relative
                                      to TopLeft of parent hitbox */
   68,10,                             /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,          /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText5, /* Item render  (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
   NULL,                                 /* alternate command-key */
   NULL,                           /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText6 = {
   3,2,JAM2,          /* front and back text pens and drawmode */
   20,1,           /* XY origin relative to container TopLeft */
   NULL,                /* font pointer or NULL for defaults */
   "      ",                                 /* pointer to text */
   NULL                             /* next IntuiText structure */
};

struct MenuItem SubItem3 = {
   &SubItem4,                          /* next SubItem structure */
   34,12,                             /* XY of Item hitbox relative
                                      to TopLeft of parent hitbox */
   68,10,                             /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,          /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText6, /* Item render  (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
   NULL,                                 /* alternate command-key */
   NULL,                           /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText7 = {
   3,1,JAM2,          /* front and back text pens and drawmode */
   20,1,           /* XY origin relative to container TopLeft */
   NULL,                /* font pointer or NULL for defaults */
   "      ",                                 /* pointer to text */
   NULL                             /* next IntuiText structure */
};

struct MenuItem SubItem2 = {
   &SubItem3,                          /* next SubItem structure */
   34,2,                              /* XY of Item hitbox relative
                                      to TopLeft of parent hitbox */
   68,10,                             /* hit box width and height */
   CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX,          /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText7,  /* Item render (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
   NULL,                                 /* alternate command-key */
   NULL,                           /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText8 = {
   3,0,JAM2,          /* front and back text pens and drawmode */
   20,1,           /* XY origin relative to container TopLeft */
```

```
    NULL,                   /* font pointer or NULL for defaults */
    "     ",                                  /* pointer to text */
    NULL                              /* next IntuiText structure */
};

struct MenuItem SubItem1 = {
    &SubItem2,                          /* next SubItem structure */
    34,-8,                             /* XY of Item hitbox relative
                                       to TopLeft of parent hitbox */
    68,10,                             /* hit box width and height */
    CHECKIT+ITEMTEXT+ITEMENABLED+HIGHBOX+CHECKED, /* Item flags */
    0,          /* each bit mutually-excludes a same-level Item */
    (APTR)&IText8,  /* Item render (IntuiText or Image or NULL) */
    NULL,                                       /* Select render */
    NULL,                                  /* alternate command-key */
    NULL,                           /* no SubItem list for SubItems */
    0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText9 = {
    3,1,COMPLEMENT,     /* front and back text pens and drawmode */
    1,1,              /* XY origin relative to container TopLeft */
    NULL,                   /* font pointer or NULL for defaults */
    "Color",                                    /* pointer to text */
    NULL                              /* next IntuiText structure */
};

struct MenuItem MenuItem2 = {
    NULL,                               /* next MenuItem structure */
    0,11,                          /* XY of Item hitbox relative to
                                       TopLeft of parent hitbox */
    49,10,                             /* hit box width and height */
    ITEMTEXT+ITEMENABLED+HIGHCOMP,                  /* Item flags */
    0,          /* each bit mutually-excludes a same-level Item */
    (APTR)&IText9,  /* Item render (IntuiText or Image or NULL) */
    NULL,                                       /* Select render */
    NULL,                                  /* alternate command-key */
    &SubItem1,                                    /* SubItem list */
    0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText10 = {
    3,1,COMPLEMENT,     /* front and back text pens and drawmode */
    1,1,              /* XY origin relative to container TopLeft */
    NULL,                   /* font pointer or NULL for defaults */
    "!!!",                                      /* pointer to text */
    NULL                              /* next IntuiText structure */
};

struct MenuItem SubItem10 = {
    NULL,                               /* next SubItem structure */
    59,0,                              /* XY of Item hitbox relative
                                       to TopLeft of parent hitbox */
    25,10,                             /* hit box width and height */
    ITEMTEXT+ITEMENABLED+HIGHCOMP,                  /* Item flags */
    0,          /* each bit mutually-excludes a same-level Item */
    (APTR)&IText10, /* Item render (IntuiText or Image or NULL) */
    NULL,                                       /* Select render */
    NULL,                                  /* alternate command-key */
    NULL,                           /* no SubItem list for SubItems */
    0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText11 = {
    3,1,COMPLEMENT,     /* front and back text pens and drawmode */
    1,1,              /* XY origin relative to container TopLeft */
    NULL,                   /* font pointer or NULL for defaults */
    "???",                                      /* pointer to text */
    NULL                              /* next IntuiText structure */
};
```

```
struct MenuItem SubItem9 = {
   &SubItem10,                          /* next SubItem structure */
   34,0,                               /* XY of Item hitbox relative
                                       to TopLeft of parent hitbox */
   25,10,                              /* hit box width and height */
   ITEMTEXT+ITEMENABLED+HIGHCOMP,                    /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText11  /* Item render (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
   NULL,                                  /* alternate command-key */
   NULL,                           /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText12 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   1,1,             /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "Pen",                                  /* pointer to text */
   NULL                               /* next IntuiText structure */
};

struct MenuItem MenuItem1 = {
   &MenuItem2,                          /* next MenuItem structure */
   0,0,                                /* XY of Item hitbox relative
                                       to TopLeft of parent hitbox */
   49,10,                              /* hit box width and height */
   ITEMTEXT+ITEMENABLED+HIGHCOMP,                    /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText12, /* Item render (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
   NULL,                                  /* alternate command-key */
   &SubItem9,                                    /* SubItem list */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct Menu Menu3 = {
   NULL,                                     /* next Menu structure */
   135,0,                                 /* XY origin of Menu hit box
                                          relative to screen TopLeft */
   66,0,                           /* Menu hit box width and height */
   MENUENABLED,                                       /* Menu flags */
   "Graphic",                                /* text of Menu name */
   &MenuItem1                    /* MenuItem linked list pointer */
};

struct IntuiText IText13 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   20,1,            /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "60 Cgaracter",                            /* pointer to text */
   NULL                               /* next IntuiText structure */
};

struct MenuItem MenuItem8 = {
   NULL,                                /* next MenuItem structure */
   0,55,                               /* XY of Item hitbox relative
                                       to TopLeft of parent hitbox */
   140,10,                             /* hit box width and height */
   CHECKIT+ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP, /*Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText13, /* Item render (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
   "6",                                   /* alternate command-key */
   NULL,                                         /* SubItem list */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText14 = {
```

```
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   20,1,            /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "80 Character",                          /* pointer to text */
   NULL                              /* next IntuiText structure */
};

struct MenuItem MenuItem7 = {
   &MenuItem8,                        /* next MenuItem structure */
   0,44,                             /* XY of Item hitbox relative
                                     to TopLeft of parent hitbox */
   140,10,                           /* hit box width and height */
   CHECKIT+ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP+CHECKED,
                                                   /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText14, /* Item render (IntuiText or Image or NULL) */
   NULL,                                      /* Select render */
   "8",                                 /* alternate command-key */
   NULL,                                      /* SubItem list */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText15 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   20,1,            /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "Invers",                                /* pointer to text */
   NULL                              /* next IntuiText structure */
};

struct MenuItem MenuItem6 = {
   &MenuItem7,                        /* next MenuItem structure */
   0,33,                             /* XY of Item hitbox relative
                                     to TopLeft of parent hitbox */
   140,10,                           /* hit box width and height */
   CHECKIT+ITEMTEXT+COMMSEQ+MENUTOGGLE+ITEMENABLED+HIGHCOMP,
                                                   /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText15, /* Item render (IntuiText or Image or NULL) */
   NULL,                                      /* Select render */
   "V",                                 /* alternate command-key */
   NULL,                                      /* SubItem list */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText16 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   20,1,            /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "Bold",                                  /* pointer to text */
   NULL                              /* next IntuiText structure */
};

struct MenuItem MenuItem5 = {
   &MenuItem6,                        /* next MenuItem structure */
   0,22,                             /* XY of Item hitbox relative
                                     to TopLeft of parent hitbox */
   140,10,                           /* hit box width and height */
   CHECKIT+ITEMTEXT+COMMSEQ+MENUTOGGLE+ITEMENABLED+HIGHCOMP,
                                                   /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText16, /* Item render (IntuiText or Image or NULL) */
   NULL,                                      /* Select render */
   "B",                                 /* alternate command-key */
   NULL,                                      /* SubItem list */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText17 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
```

```
    20,1,           /* XY origin relative to container TopLeft */
    NULL,               /* font pointer or NULL for defaults */
    "Italic",                           /* pointer to text */
    NULL                            /* next IntuiText structure */
};

struct MenuItem MenuItem4 = {
    &MenuItem5,                      /* next MenuItem structure */
    0,11,                           /* XY of Item hitbox relative
                                    to TopLeft of parent hitbox */
    140,10,                         /* hit box width and height */
    CHECKIT+ITEMTEXT+COMMSEQ+MENUTOGGLE+ITEMENABLED+HIGHCOMP,
                                                    /* Item flags */
    0,          /* each bit mutually-excludes a same-level Item */
    (APTR)&IText17, /* Item render (IntuiText or Image or NULL) */
    NULL,                                   /* Select render */
    "I",                                /* alternate command-key */
    NULL,                                    /* SubItem list */
    0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText18 = {
    3,1,COMPLEMENT,     /* front and back text pens and drawmode */
    20,1,           /* XY origin relative to container TopLeft */
    NULL,               /* font pointer or NULL for defaults */
    "Normal",                           /* pointer to text */
    NULL                            /* next IntuiText structure */
};

struct MenuItem MenuItem3 = {
    &MenuItem4,                      /* next MenuItem structure */
    0,0,                            /* XY of Item hitbox relative
                                    to TopLeft of parent hitbox */
    140,10,                         /* hit box width and height */
 CHECKIT+ITEMTEXT+COMMSEQ+MENUTOGGLE+ITEMENABLED+HIGHCOMP+CHECKED,
                                                    /* Item flags */
    0,          /* each bit mutually-excludes a same-level Item */
    (APTR)&IText18, /* Item render (IntuiText or Image or NULL) */
    NULL,                                   /* Select render */
    "N",                                /* alternate command-key */
    NULL,                                    /* SubItem list */
    0xFFFF        /* filled in by Intuition for drag selections */
};

struct Menu Menu2 = {
    &Menu3,                             /* next Menu structure */
    63,0,                           /* XY origin of Menu hit box
                                    relative to screen TopLeft */
    66,0,                       /* Menu hit box width and height */
    MENUENABLED,                                /* Menu flags */
    "Style",                             /* text of Menu name */
    &MenuItem3                  /* MenuItem linked list pointer */
};

struct IntuiText IText19 = {
    3,1,COMPLEMENT,     /* front and back text pens and drawmode */
    1,1,            /* XY origin relative to container TopLeft */
    NULL,               /* font pointer or NULL for defaults */
    "Program end",                          /* pointer to text */
    NULL                            /* next IntuiText structure */
};

struct MenuItem MenuItem12 = {
    NULL,                            /* next MenuItem structure */
    0,33,                           /* XY of Item hitbox relative
                                    to TopLeft of parent hitbox */
    137,10,                         /* hit box width and height */
    ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,        /* Item flags */
    0,          /* each bit mutually-excludes a same-level Item */
    (APTR)&IText19, /* Item render (IntuiText or Image or NULL) */
```

```
   NULL,                                        /* Select render */
   "E",                                  /* alternate command-key */
   NULL,                                         /* SubItem list */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText20 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   1,1,             /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "Delete",                              /* pointer to text */
   NULL                             /* next IntuiText structure */
};

struct MenuItem MenuItem11 = {
   &MenuItem12,                      /* next MenuItem structure */
   0,22,                            /* XY of Item hitbox relative
                                    to TopLeft of parent hitbox */
   137,10,                          /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,        /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText20, /* Item render (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
   "D",                                  /* alternate command-key */
   NULL,                                         /* SubItem list */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText21 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   1,1,             /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "IFF",                                 /* pointer to text */
   NULL                             /* next IntuiText structure */
};

struct MenuItem SubItem13 = {
   NULL,                             /* next SubItem structure */
   122,12,                          /* XY of Item hitbox relative
                                    to TopLeft of parent hitbox */
   81,10,                           /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,        /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText21, /* Item render (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
   "F",                                 /* alternate command-key */
   NULL,                       /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText22 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   1,1,             /* XY origin relative to container TopLeft */
   NULL,                 /* font pointer or NULL for defaults */
   "ASCII",                               /* pointer to text */
   NULL                             /* next IntuiText structure */
};

struct MenuItem SubItem12 = {
   &SubItem13,                       /* next SubItem structure */
   122,2,                           /* XY of Item hitbox relative
                                    to TopLeft of parent hitbox */
   81,10,                           /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,        /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText22, /* Item render (IntuiText or Image or NULL) */
   NULL,                                        /* Select render */
   "C",                                 /* alternate command-key */
   NULL,                       /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
```

```
};

struct IntuiText IText23 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
   1,1,           /* XY origin relative to container TopLeft */
   NULL,               /* font pointer or NULL for defaults */
   "Text",                                /* pointer to text */
   NULL                           /* next IntuiText structure */
};

struct MenuItem SubItem11 = {
   &SubItem12,                          /* next SubItem structure */
   122,-8,                           /* XY of Item hitbox relative
                                     to TopLeft of parent hitbox */
   81,10,                            /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,        /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText23, /* Item render (IntuiText or Image or NULL) */
   NULL,                                      /* Select render */
   "X",                                /* alternate command-key */
   NULL,                        /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText24 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   1,1,            /* XY origin relative to container TopLeft */
   NULL,               /* font pointer or NULL for defaults */
   "Save",                             /* pointer to text */
   NULL                           /* next IntuiText structure */
};

struct MenuItem MenuItem10 = {
   &MenuItem11,                         /* next MenuItem structure */
   0,11,                             /* XY of Item hitbox relative
                                     to TopLeft of parent hitbox */
   137,10,                           /* hit box width and height */
   ITEMTEXT+ITEMENABLED+HIGHCOMP,                /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText24, /* Item render (IntuiText or Image or NULL) */
   NULL,                                      /* Select render */
   NULL,                                /* alternate command-key */
   &SubItem11,                                 /* SubItem list */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText25 = {
   3,1,COMPLEMENT,    /* front and back text pens and drawmode */
   1,1,           /* XY origin relative to container TopLeft */
   NULL,               /* font pointer or NULL for defaults */
   "IFF",                                 /* pointer to text */
   NULL                           /* next IntuiText structure */
};

struct MenuItem SubItem16 = {
   NULL,                                /* next SubItem structure */
   122,12,                           /* XY of Item hitbox relative
                                     to TopLeft of parent hitbox */
   81,10,                            /* hit box width and height */
   ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,        /* Item flags */
   0,          /* each bit mutually-excludes a same-level Item */
   (APTR)&IText25, /* Item render (IntuiText or Image or NULL) */
   NULL,                                      /* Select render */
   "I",                                /* alternate command-key */
   NULL,                        /* no SubItem list for SubItems */
   0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText26 = {
   3,1,COMPLEMENT,   /* front and back text pens and drawmode */
```

```
    1,1,            /* XY origin relative to container TopLeft */
    NULL,                /* font pointer or NULL for defaults */
    "ASCII",                                /* pointer to text */
    NULL                            /* next IntuiText structure */
};

struct MenuItem SubItem15 = {
    &SubItem16,                          /* next SubItem structure */
    122,2,                              /* XY of Item hitbox relative
                                        to TopLeft of parent hitbox */
    81,10,                              /* hit box width and height */
    ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,        /* Item flags */
    0,          /* each bit mutually-excludes a same-level Item */
    (APTR)&IText26, /* Item render (IntuiText or Image or NULL) */
    NULL,                                       /* Select render */
    "A",                                 /* alternate command-key */
    NULL,                           /* no SubItem list for SubItems */
    0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText27 = {
    3,1,COMPLEMENT,     /* front and back text pens and drawmode */
    1,1,              /* XY origin relative to container TopLeft */
    NULL,                /* font pointer or NULL for defaults */
    "Text",                                 /* pointer to text */
    NULL                              /* next IntuiText structure */
};

struct MenuItem SubItem14 = {
    &SubItem15,                          /* next SubItem structure */
    122,-8,                             /* XY of Item hitbox relative
                                       to TopLeft of parent hitbox */
    81,10,                              /* hit box width and height */
    ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,        /* Item flags */
    0,          /* each bit mutually-excludes a same-level Item */
    (APTR)&IText27, /* Item render (IntuiText or Image or NULL) */
    NULL,                                       /* Select render */
    "T",                                 /* alternate command-key */
    NULL,                           /* no SubItem list for SubItems */
    0xFFFF        /* filled in by Intuition for drag selections */
};

struct IntuiText IText28 = {
    3,1,COMPLEMENT,     /* front and back text pens and drawmode */
    1,1,              /* XY origin relative to container TopLeft */
    NULL,                /* font pointer or NULL for defaults */
    "Laden",                                /* pointer to text */
    NULL                              /* next IntuiText structure */
};

struct MenuItem MenuItem9 = {
    &MenuItem10,                         /* next MenuItem structure */
    0,0,                                /* XY of Item hitbox relative
                                       to TopLeft of parent hitbox */
    137,10,                             /* hit box width and height */
    ITEMTEXT+ITEMENABLED+HIGHCOMP,                /* Item flags */
    0,          /* each bit mutually-excludes a same-level Item */
    (APTR)&IText28, /* Item render (IntuiText or Image or NULL) */
    NULL,                                       /* Select render */
    NULL,                                /* alternate command-key */
    &SubItem14,                                  /* SubItem list */
    0xFFFF        /* filled in by Intuition for drag selections */
};

struct Menu Menu1 = {
    &Menu2,                                 /* next Menu structure */
    0,0,                               /* XY origin of Menu hit box
                                       relative to screen TopLeft */
    57,0,                         /* Menu hit box width and height */
    MENUENABLED,                                /* Menu flags */
```

```
   "Datei",                              /* text of Menu name */
   &MenuItem9                    /* MenuItem linked list pointer */
};

#define MenuList Menu1

struct NewWindow NewWindowStructure = {
   275,85,   /* window XY origin relative to TopLeft of screen */
   150,50,                          /* window width and height */
   0,1,                              /* detail and block pens */
   NULL,                                      /* IDCMP flags */
   NULL,                                 /* other window flags */
   NULL,                          /* first gadget in gadget list */
   NULL,                             /* custom CHECKMARK imagery */
   "Your new window",                           /* window title */
   NULL,                                     /* custom screen */
   NULL,                                     /* custom bitmap */
   5,5,                             /* minimum width and height */
   640,200,                         /* maximum width and height */
   WBENCHSCREEN                      /* destination screen type */
};

/* end of PowerWindows source generation */
```

3.8.4.4 Fine-tuning the source text

The source text created by PowerWindows® is very long, granted, but this is the way PowerWindows works. Unfortunately you can't tell the program to generate a structure with all its data in one line.

Let's close this section by looking at how you can further process the source text. In addition to the menu strip, PowerWindows creates a `NewWindow` structure for us.

It always defines an `IntuiText` structure together with a `MenuItem` structure. These are numbered for some inexplicable reasons. If you would like to add to the program, you must change the numbers. Unfortunately, these numbers don't give much information. It would be better to have the names of the menu items there. You can add these to make the insertion of new menu items easier.

The structures of the program particularly stress the importance of having menu null arguments. Unfortunately, you may forget which meaning refers to what. Also, you must change the structure coordinate choices to make room.

The graphics must be inserted in the first submenu item of the third menu (`Graphic`). Remove the `IntuiText` structures and insert the `Image` structures (remember to remove the `ITEMNEXT` flag).

Note: We advise that you make as few changes as possible, since it is easy to insert an error—and this can be very hard to find later on. The simplest method is to print out the entire program text and then mark the changed places with a marker. If the operation is successful, you can

make the new changes. But always do it between compiling, so you have the results before your eyes.

3.9 The Console Device

The IDCMP sometimes only allows the receiving of data. You can select which data should be transmitted, but the receiving of data is limited.

The IDCMP uses two methods to read the keyboard. RAWKEY always sends the keyboard code without changing it to the keyboard table form (that's why the word RAW in RAWKEY). VANILLAKEY translates the keyboard code according to the current keyboard table. This gives us the power to change the keyboard to another country's keyboard layout.

We still have a problem—no data can be sent. We must rely on the console.device to make this possible. The console.device is a device with which we can exchange data. It receives data from the IDCMP through RAWKEY as well as VANILLAKEY. In addition, the console device processes output. It displays text according to the window's size. This ensures that no word is cut off at the end of a line. The console.device also scrolls the text display, processes text graphically and creates messages under certain conditions.

3.9.1 Designing Communication Direction

We need a read and write port for the console.device as we would for any device. These ports then allow us to communicate using IOStandardMessages. To set the ports and the standard input/output, we need two functions that can act with Exec. The Open_All() routine calls CreatePort() and CreateStdIO():

```
ConsoleWritePort = CreatePort("wgb-con-dev.write", 0L);
if (!ConsoleWritePort)
   {
   printf("The new WritePort could not be enabled!\n");
   Close_All();
   exit(FALSE);
   }

ConsoleWriteMsg  = CreateStdIO(ConsoleWritePort);
if (!ConsoleWriteMsg)
   {
   printf("CreateStdIO for Write not enabled!\n");
   Close_All();
   exit(FALSE);
   }
```

The same general process holds true for reading data:

```
ConsoleReadPort = CreatePort("wgb-con-dev.read", 0L);
if (!ConsoleReadPort)
   {
   printf("The ReadPort could not be enabled!\n");
   Close_All();
   exit(FALSE);
   }

ConsoleReadMsg  = CreateStdIO(ConsoleReadPort);
if (!ConsoleReadMsg)
   {
   printf("CreateStdIO for Read not enabled!\n");
   Close_All();
   exit(FALSE);
   }
```

Once these two data directions are enabled, the device can be opened. Two fields must be initialized in the `IOStdRequest` structure. This is unusual when opening a device, but the console device needs this initialization:

```
ConsoleWriteMsg->io_Data   = (APTR)FirstWindow;
ConsoleWriteMsg->io_Length = sizeof(*FirstWindow);

conDevErr = OpenDevice ("Console.device", 0L,
                                    ConsoleWriteMsg, 0L);
if (conDevErr)
   {
   printf("OpenDevice error!\n");
   Close_All();
   exit(FALSE);
   };

ConsoleReadMsg->io_Device  = ConsoleWriteMsg->io_Device;
ConsoleReadMsg->io_Unit    = ConsoleWriteMsg->io_Unit;
```

The `ConsoleWriteMsg` structure is attached to a window over which the data exchange should go. After opening the window, the `ConsoleReadMsg` structure receives the device and unit message. Now the input and output can begin.

All of the keyboard input going to the specified window is first directed to and processed by the console.device. The program then receives a report that contains the translated keyboard codes from the console.device.

If you want to send messages to the console.device, then you use the `ConsoleWriteMsg` structure. This structure directs commands and instructions to the device.

3.9.2 Receiving the First Data

We don't need to do anything extra in data direction to receive data. We're ready to receive data from the keyboard. But how does it run?

The message must be sent to the console.device concerning a specific amount of keyboard input. A `StandardRequest` structure sends this instruction. The program doesn't wait until it receives an answer, because other actions can occur within that time.

It would be simplest if you wrote a function for the command to read keys. This function could have a name of `QueueRead()`

```
/*************************************
 *                                   *
 * Function:  Input over ConsoleDev  *
 * ================================= *
 * Author:  Date:      comments:     *
 * ------  ----------  ----------    *
 * Wgb     24.10.1987  improved L&D  *
 * StdReq   pointer to IOStdReq      *
 * buffer   Address of  Text buffer  *
 * length   Length of Text buffer    *
 *************************************/

QueueRead(StdReq, buffer, Length)

struct IOStdReq *StdReq;
char            *buffer;
ULONG            length;

   {
   StdReq->io_Command = CMD_READ;
   StdReq->io_Data    = (APTR)buffer;
   StdReq->io_Length  = length;
   SendIO(StdReq);
   }
```

The function assigns the pointer to the `IOStdReq` structure which we have decided to write to. You also assign the pointer to a buffer where the received data will be stored later, and conclude with the buffer's length (i.e., the number of characters that should be received):

```
QueueRead(consoleReadMsg, &Buffer[0], 1L);
```

Now our loop can be on standby until `GetMsg(ConsoleReadPort)` receives a response . Then a new reading routine is used:

```
if (GetMsg(consoleReadPort))
     Write(consoleWriteMag, &[0], 1L);
```

3.9.3 Displaying the characters

It would be better if we tested the characters for command codes before output, but we'll take the easy way out and redisplay the characters on the console.

You need a function that works basically like the `read` function: An `IOStdReq` is filled, the `write` option is chosen, the text buffer transfers, and the length is added. Then the message is sent. The routine waits until execution ends, to avoid any overlap:

```
/*************************************
 *                                   *
 * Functions: Output over ConsoleDev *
 * ================================= *
 *                                   *
 * Author:  Date:     comments:      *
 * ------  ----------  ----------     *
 * Wgb     23.10.1987  Basic L&D      *
 *                                   *
 * StdReq   Message port of the report *
 * Zchar    Number of characters     *
 *                                   *
 *************************************/


ConWrite(StdReq, Text, Length)

struct IOStdReq *StdReq;
char             Text[];
int              Length;

   {
   StdReq->io_Command = CMD_WRITE;
   StdReq->io_Data    = (APTR)Text;
   StdReq->io_Length  = Length;
   DoIO(StdReq);
   }
```

This function transfers the characters at the start of the buffer when called. We can also display a string terminated by a null byte.

Here you use the length descriptor -1L. To test this, have the `intuition.library` open a window and create a loop which checks for a clicked close gadget. A console.device window opens featuring complete input and output, resulting in a full window editor. Check it out!

3.9.4 Console Device Control Sequences

Unlike `IntuiText` structures, console parameters may be changed, enabled and disabled through the console.device. These control sequences consist of simple character codes or longer character strings. You transmit the necessary text to execute the control sequence.

The first table of control characters below perform very simple tasks:

Command	Character sequence	Comments
BACKSPACE	0x08	Moves cursor one column left
LINE FEED	0x0A	Moves cursor one line down to next line depending on the mode function (usually to next line but not to the first column)
VERTICAL TAB	0x0B	Moves cursor one line up
FORM FEED	0x0C	Erases contents of window
CARRIAGE RETURN	0x0D	Moves cursor to first text column (but not one line down)
SHIFT IN	0x0E	Disables SHIFT OUT (text mode)
SHIFT OUT	0x0F	Enables SHIFT OUT (graphic mode—sets bit 7 of all character codes)
ESC	0x1B	Enables escape sequences (used mostly with printed output)
CSI	0x9B	control sequence introducer-enables control sequences (used mostly with the console device)

The last two characters mentioned above are very important. CSI introduces a control sequence; <Esc> performs a similar task:

Label	Character Sequence	Comments
RESET	Esc 0x63	Returns to original status
INSERT [N] CHARACTERS	CSI [N] 0x40	Inserts [N] characters. If no [N] is given, N = 1
CURSOR UP [N] LINES	CSI [N] 0x41	Moves cursor [N] lines up. If no [N] is given, N = 1
CURSOR DOWN [N] LINES	CSI [N] 0x42	Moves cursor [N] lines down. If no [N] is given, N = 1
CURSOR FORWARD	CSI [N] 0x44	Moves cursor [N] characters to [N] CHARACTERS to the right. If no [N] is given, N = 1

Label	Character Sequence	Comments
CURSOR BACKWARD	CSI [N] 0x44	Moves cursor [N] characters to [N] CHARACTERS to the right. If no [N] is given, N = 1
CURSOR NEXT LINE [N]	CSI [N] 0x45	Moves the cursor to line [N]. If no [N] is given, N = 1
CURSOR PRECEDING	CSI [N] 0x46	Moves cursor to previous [N] LINE [N] line. If no [N] is given, N = 1
MOVE CURSOR TO	CSI [N] (0x3B [M]) 0x48	Moves cursor to line [N], ROW [N] (COLUMN [M]) (column [M])
ERASE TO END OF	CSI 0x4A	Erases window from current DISPLAY cursor position to end
ERASE TO END OF LINE	CSI 0x4B	Deletes window to end of line
INSERT LINE	CSI 0x4C	Inserts a line
DELETE LINE	CSI 0x4D	Deletes a line
DELETE CHARACTER [N]	CSI [N] 0x50	Deletes characters to right of cursor
SCROLL UP [N] LINES	CSI [N] 0x53	Scrolls text [N] lines up
SCROLL DOWN [N] LINES	CSI [N] 0x54	Scrolls text [N] lines down
SET MODE	CSI 0x32 0x30 0x68	Changes linefeed to carriage return/linefeed
RESET MODE	CSI 0x32 0x30 0x6C	Changes linefeed to linefeed only (no added carriage return)
DEVICE STATUS REPORT	CSI 0x36 0x6E	Requests status report

Some applications allow combined control sequences. Multiple sequences differ from single sequences in that they consist of multiple groups of sequences. Another difference in multiple sequences: One number performs the multiple functions. This number appears as an ASCII number, rather than as a series of ASCII codes (the usual method).

You can eliminate the use of numbers here. The console takes a standard value given in the table. Many of the command sequences call upon the console.device to return a message containing the requested data (e.g., cursor position).

We suggest that you create an array from which you can recall a sequence when you need it. This is because all of the command sequences in the console.device consist of multiple characters. The routine for reading the characters is sensitive to differences between letters, numbers and control characters. Control characters are divided into other groups. Here you need additional routines to read cursor keys, function keys and the control sequences (which return to the user) directed by the CSI. Here is an example of important control characters and a command table:

```
#define SPACE                  0x20
#define BACKSPACE              0x08
#define LINEFEED               0x0A
#define RETURN                 0x0D
#define CSI                    0x9B
#define DEVICE_STATUS_REPORT   0x36, 0x6E
#define WINDOW_STATUS_REQUEST  0x30, 0x20, 0x71

#define CURSOR_UP              0x41
#define CURSOR_DOWN            0x42
#define CURSOR_Shift_UP        0x54
#define CURSOR_Shift_DOWN      0x53


UBYTE Specialcharacter[] =
   {
   SPACE,
   BACKSPACE,
   LINEFEED,
   RETURN,
   CSI, DEVICE_STATUS_REPORT,
   CSI, WINDOW_STATUS_REQUEST
   };
```

3.9.5 Reading Messages from the Console

The system of communication is almost ready. We can now transmit and receive data over two ports. We know which "normal" characters can be received and which control sequences are available. However, the program still needs a few details before it can handle data.

Receiving simple keystrokes is no problem for the program. However, messages consisting of multiple characters can make things more complicated. Also there is danger that the program will misinterpret the message or process the characters incorrectly.

We want to avoid these errors from the beginning and create a routine that evaluates everything correctly. Since our first concern here is with reading, we should make sure that a large enough text buffer exists to

handle the entire window. We need a reading loop that takes data from our data buffer:

```
if (GetMsg(consoleReadPort))
   {
   switch (Buffer[0])
      {
         /* Selection */
      }
   QueueRead(consoleReadMsg, &Buffer[mode], 1L);
   }
```

As always, the reading loop has a primitive assignment: See if data is present, select the first character, select a new character, etc. The routine treats the character as a letter or normal character that doesn't need special handling. Then the following lines come into play:

```
default        : ConWriteChr(consoleWriteMsg, &Buffer[0]);
                 Coords = FALSE;
                 WindowBuffer[y][x-1] = Buffer[0];
                 if (WindowBuffer[y][79] < x)
                     WindowBuffer[y][79] = x;
                 break;
```

The small processing routine above gets the first character. Then the coordinate flag changes to FALSE, so the program knows that the actual coordinates are no longer known. The characters are then transferred to the window buffer. This stores all of the characters presently in the window. Next the new character is tested to see if it exceeds the current line length. If so, the new line length is transferred.

These methods were chosen for the following purposes: The window buffer clears at the beginning, so that it contains only spaces (0x20). When you enter <Return> in a line, this should be sent to DOS so the CLI command can be executed. If all the spaces are transferred to DOS, then the CLI cannot recognize where the command sequence ends. You can't send a command because the CLI command word and arguments cannot be transmitted with the added characters.

Imagine a line of command data. The line begins with the first character that must be examined. Then the line in which the cursor is found should be transmitted:

```
case RETURN    : ConWriteChr(consoleWriteMsg, &Buffer[0])
      ConWriteChr(consoleWriteMsg, &Specialcharacter[2]);
                 Coords = FALSE;
                 if (WindowBuffer[y][79] != 0)
                    {
              WindowBuffer[y][WindowBuffer[y][79]] = NULL;
              Execute(&WindowBuffer[y][0], 0, 0);
              WindowBuffer[y][WindowBuffer[y][78]] = 0x20;
                    }
                  break;
```

The second character reading routine processes the graphic first. This means that the routine requires cursor placement at the beginning of the line and in the next line. The coordinate flag is set again. A test examines whether characters were entered in the line. If so, a null ends the line so the `Execute()` function of DOS can be called. After that a space ends the line.

The last character needing special attention is the <Backspace> key. Neither the console.device task nor the buffer correction should execute:

```
case BACKSPACE : ConWriteChr(consoleWriteMsg,
&Buffer[0]);
      ConWrite(consoleWriteMsg, &Specialcharacter[0], 2);
                WindowBuffer[y][x-1] = SPACE;
                Coords = FALSE;
                break;
```

The last and most important problem (control sequence management) is delayed. Look at the following short form:

```
case CSI     : mode += 1;
               break
```

This read loop assigns a number that increments by one every time further readings occur. From the number you can recognize how many characters must still be processed. Multiple control sequences may be in the buffer, because the CSI controls the function keys and cursor keys. We will look at the same reader:

```
/*********************************
 *                               *
 *  Handling of the control      *
 *  sequences                    *
 *                               *
 *********************************/

else if (mode)
   {
       /* Check */

   mode = 0;
   QueueRead(consoleReadMsg, &Buffer[mode], 1L);
   }
```

The `ELSE` belongs to the `IF` statement that tests the console buffer for a character from the device. As long as there is a message there, this gets the data first from the buffer because the console.device only stores the last 32 characters received. If more characters exist, the routine simply ignores them. This is why the variable `mode` is used; to enter the new character in the next location of our buffer.

Let's look at the handling of the first control sequence, which reports the cursor position:

```
/*********************************
 *                               *
 *  Control Sequence :           *
 *                               *
 *  CURSOR POSITION REPORT       *
 *                               *
 *********************************/

if (Buffer[mode-1] == 82)
   {
   Coords = TRUE;
   x = 0; y = 0; i = 1;

   while (Buffer[i] < 0x3A & 0x2F < Buffer[i])
      {
      y *= 10;
      y += Buffer[i] - 0x30;
      i ++;
      }
   i ++;
   while (Buffer[i] < 0x3A & 0x2F < Buffer[i])
      {
      x *= 10;
      x += Buffer[i] - 0x30;
      i ++;
      }
   }
```

This program section goes through one character after another, and calculates the X and Y coordinates of the cursor. The DEVICE STATUS REPORT has the format (note the ending R):

```
CSI Row; Line R
```

Next the routine evaluates the buffer character for character. The calculation of the first value concludes when the program encounters a semicolon. Then the X value can be calculated. It is also important that the coordinate flag be reset. Later a test follows that sends the set flag to the console as a report, that should include the coordinates in the data direction. This could provide us with an endless communication system consisting of the cursor coordinates only.

The next test shows that a control sequence is present (but not the DEVICE STATUS REPORT). It is first checked to see if the cursor keys were pressed. The console.device allows the use of <Shift> and cursor keys. A <Shift>ed cursor key scrolls the entire window, instead of simply changing the cursor position. This scrolling must also occur in the text buffer.

Look at the following routine:

```
else
   {
   switch (Buffer[mode-1])
      {

      /*********************************
       *                               *
       *  Control Sequence :           *
       *                               *
       *  SHIFTED CURSOR MOVE          *
       *                               *
       *********************************/

      case CURSOR_Shift_UP     : Move_Down();
                                 break;
      case CURSOR_Shift_DOWN   : Move_Up();
                                 break;

      /*********************************
       *                               *
       *  Control Sequence :           *
       *                               *
       *  NORMAL CURSOR MOVE           *
       *                               *
       *********************************/

      case CURSOR_UP           : if (y == 1) Move_Down();
                                 break;
      case CURSOR_DOWN         : if (y == Ymax) Move_Up();
                                 break;
      }
```

The last two statements end with normal cursor key movements. If the cursor has reached the top or bottom border of the screen, the screen scrolls up or down by one line.

Two functions handle the scrolling of the window buffer. These functions look like this:

```
/*************************************
 *                                   *
 * Functions: Text buffer processing *
 * ================================= *
 *                                   *
 * Author:  Date:      comments:     *
 * ------  ----------  ----------    *
 * Wgb      11/2/1987                *
 *                                   *
 * y  Number of the line             *
 * x  Number of the column           *
 *                                   *
 *************************************/


Clear_Buffer()
```

```
   {
   int y;
   for(y=0; y<80; y ++)
      Clear_Line(y);
   }


Clear_Line(y)
int y;
   {
   int x;
   for(x=0; x<79; x ++)
      WindowBuffer[y][x] = 0x20;
   WindowBuffer[y][78] = NULL;       /* Line end mark */
   WindowBuffer[y][79] = NULL; /* Number of entered
characters */
   }


Move_Up()
   {
   int x, y;
   for(y=1; y<80; y ++)
      for(x=0; x<80; x ++)
         WindowBuffer[y-1][x] = WindowBuffer[y][x];
   Clear_Line(79);
   }


Move_Down()
   {
   int x, y;
   for(y=79; y>0; y --)
      for(x=0; x<80; x ++)
         WindowBuffer[y][x] = WindowBuffer[y-1][x];
   Clear_Line(0);
   }
```

Two functions taken from above are of particular interest: `Clear_Buffer()` and `Clear_Line()`.

`Clear_Buffer()`

This function initializes the buffer at the beginning of the code. The number of characters entered per line is set at zero, just as a zero indicates the end of a line of text.

`Clear_Line()`

The `Clear_Buffer()` function calls this function. It allows the deletion of selected lines supported by control sequences.

It would take far too long to describe the handling and testing of every control sequence in this chapter. However, the next routine is a generic loop that assigns any routines that aren't handled to output status:

```
/*********************************
 *                               *
 *  Control Sequence :           *
 *                               *
 *Manipulation of any sequence   *
 *                               *
 *********************************/

   ConWrite(consoleWriteMsg, &Buffer[0], mode);
   Coords = FALSE;
   }
```

You can enter all of the additional control sequence functions, if you want to. For example, you won't find a description of simple function key assignment. It's actually very simple. Assuming that you want these assignments permanent within the program, then you simply establish a pointer array and add the desired texts. Here is an example for our CLI editor:

```
UBYTE *Functionkeys[] =
   {
   "dir"+RETURN,
   "cd df1:"+RETURN,
   "cd sys:"+RETURN,
   "list"+RETURN,
   "makedir ram:c"+RETURN+"copy sys:c ram:c"+RETURN,
   "execute make "
   };
```

The test for the function keys must identify these through "~" and the number calculated. Then the pointer can be assigned to ConWrite();:

```
ConWrite(consoleWriteMasg, Functionkeys[Number], -1L);
```

3.9.6 Finishing the CLI Editor

When operating between Intuition, the editor window and the console, we recommend that you open the console device through some other method. We would like to describe that briefly here:

Although the console.device is built into the IDCMP, there are still more options available. For total openness in manipulation we first define an IOStdReq structure, in which we set a pointer to our window in the value io_Data. The device can be opened as usual. Now it is connected to our Intuition window, and communication can be carried over to the structure in use. They can be used to write as well as read. The read and write functions that were developed at the beginning of this chapter serve the same purpose as before.

Set all of the reading elements, all of the routines to open and close, and the text buffer manipulator (remember to define the important variables and arrays) Now we have a new CLI editor!

Note: The window sizing gadgets described earlier in this book can be added to this editor. Here is a complete listing:

```
/*************************************
 *                                   *
 * Program: New CLI-Ed  workings     *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/20/1987  for testing only *
 *                                   *
 *                                   *
 *************************************/


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

struct IntuitionBase *IntuitionBase;
struct Window        *FirstWindow;
struct IntuiMessage  *message;

struct IOStdReq *consoleWriteMsg,  *consoleReadMsg,
*CreateStdIO();
struct MsgPort  *consoleWritePort, *consoleReadPort,
*CreatePort();

struct NewWindow ConsoleWindow =
   {
   0, 0,                /* LeftEdge, TopEdge   */
   640, 101,            /* Width, Height       */
   2, 3,                /* DetailPen, BlockPen */
   CLOSEWINDOW |        /* IDCMP Flags         */
   GADGETUP,
   WINDOWDEPTH |        /* Flags               */
   WINDOWSIZING |
   WINDOWDRAG |
   WINDOWCLOSE |
   ACTIVATE |
   SMART_REFRESH,
   NULL,                /* First Gadget        */
   NULL,                /* CheckMark           */
   (UBYTE *)"Wgb Prod. presents  BECKERshell",
   NULL,                /* Screen              */
   NULL,                /* BitMap              */
   640, 50,             /* Min Width, Height   */
   640, 200,            /* Max Width, Height   */
   WBENCHSCREEN,        /* Type                */
   };
```

```
#define SPACE                   0x20
#define BACKSPACE               0x08
#define LINEFEED                0x0a
#define RETURN                  0x0d
#define CSI                     0x9b
#define DEVICE_STATUS_REPORT    0x36, 0x6e
#define WINDOW_STATUS_REQUEST   0x30, 0x20, 0x71

#define CURSOR_UP               0x41
#define CURSOR_DOWN             0x42
#define CURSOR_Shift_UP         0x54
#define CURSOR_Shift_DOWN       0x53


UBYTE Specialcharacter[] =
   {
   SPACE,
   BACKSPACE,
   LINEFEED,
   RETURN,
   CSI, DEVICE_STATUS_REPORT,
   CSI, WINDOW_STATUS_REQUEST
   };


SHORT conDevErr = FALSE;
UBYTE WindowBuffer[80][80];
UBYTE Buffer[80];


VOID main()
   {
   ULONG MessageClass;
   USHORT code;
   int i;
   int x, y;
   int Xmax = 77, Ymax = 11;
   int mode = 0;
   BOOL Coords;

   struct Message *GetMsg();

   Open_All();

   Clear_Buffer();

   ConWrite(consoleWriteMsg, &Specialcharacter[4], 3);
   QueueRead(consoleReadMsg, &Buffer[0], 1L);

   FOREVER
      {
      if (message = (struct IntuiMessage *)
          GetMsg(FirstWindow->UserPort))
         {
         MessageClass = message->Class;
```

```
            code = message->Code;
            ReplyMsg(message);
            switch (MessageClass)
               {
               case CLOSEWINDOW : Close_All();
                                  exit(TRUE);
                                  break;
               }
            }
         if (GetMsg(consoleReadPort))
            {
            switch (Buffer[0])
               {
               case BACKSPACE : ConWriteChr(consoleWriteMsg,
&Buffer[0]);
                                ConWrite(consoleWriteMsg,
&Specialcharacter[0], 2);
                                Coords = FALSE;
                                break;

               case RETURN    : ConWriteChr(consoleWriteMsg,
&Buffer[0]);
                                ConWriteChr(consoleWriteMsg,
&Specialcharacter[2]);
                                Coords = FALSE;
                                if (WindowBuffer[y][79] != 0)

                                   {

WindowBuffer[y][WindowBuffer[y][79]] = NULL;
                                   Execute(&WindowBuffer[y][0], 0,
0);

WindowBuffer[y][WindowBuffer[y][79]] = 0x20;
                                   }
                                break;

               case CSI       : mode += 1;
                                break;

               default        : ConWriteChr(consoleWriteMsg,
&Buffer[0]);
                                Coords = FALSE;
                                WindowBuffer[y][x-1] = Buffer[0];

                                if (WindowBuffer[y][79] < x)
                                   WindowBuffer[y][79] = x;
                                break;
               }
            QueueRead(consoleReadMsg, &Buffer[mode], 1L);
            }


         /*********************************
          *                               *
          *  Handle the Control-          *
```

```
 *  sequence                    *
 *                              *
 *******************************/

else if (mode)
   {
   /*******************************
    *                             *
    *  Control Sequence :         *
    *                             *
    *  CURSOR POSITION REPORT     *
    *                             *
    *******************************/

   if (Buffer[mode-1] == 82)
      {
      Coords = TRUE;
      x = 0; y = 0; i = 1;

      while (Buffer[i] < 0x3a & 0x2f < Buffer[i])
         {
         y *= 10;
         y += Buffer[i] - 0x30;
         i ++;
         }
      i ++;
      while (Buffer[i] < 0x3a & 0x2f < Buffer[i])
         {
         x *= 10;
         x += Buffer[i] - 0x30;
         i ++;
         }
      }

   else
      {
      switch (Buffer[mode-1])
         {

         /*******************************
          *                             *
          *  Control Sequence :         *
          *                             *
          *  SHIFTED CURSOR MOVE        *
          *                             *
          *******************************/

         case CURSOR_Shift_UP    : Move_Down();
                                   break;
         case CURSOR_Shift_DOWN  : Move_Up();
                                   break;

         /*******************************
          *                             *
          *  Control Sequence :         *
          *                             *
```

```
                    *  NORMAL CURSOR MOVE          *
                    *                              *
                    ********************************/

                   case CURSOR_UP          : if (y == 1) Move_Down();

                                             break;
                   case CURSOR_DOWN        : if (y == Ymax)
Move_Up();
                                             break;
                   }

               /*********************************
                *                               *
                *  Control Sequence :           *
                *                               *
                *  Manipulation of any sequence *
                *                               *
                *********************************/

                  ConWrite(consoleWriteMsg, &Buffer[0], mode);
                  Coords = FALSE;
                  }


               /*********************************
                *                               *
                *  Console Device :             *
                *                               *
                *  New character read           *
                *                               *
                *********************************/

               mode = 0;
               QueueRead(consoleReadMsg, &Buffer[mode], 1L);
               }


            /*********************************
             *                              *
             *  Console Device :            *
             *                              *
             *  New cursor-                 *
             *  position                    *
             *                              *
             *********************************/

            if (Coords == FALSE)
               {
               ConWrite(consoleWriteMsg, &Specialcharacter[4], 3);

               Coords = TRUE;
               }
            }
         }
```

```
/*************************************
 *                                   *
 * Function: open all                *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/20/1987                *
 *                                   *
 * no parameter                      *
 *                                   *
 *************************************/


Open_All()

   {
   struct Library *OpenLibrary();
   struct Window  *OpenWindow();

   if (!(IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library", 0L)))
      {
      printf("Intuition Library not found!\n");
      Close_All();
      exit(FALSE);
      }

   if (!(FirstWindow = (struct Window *)
       OpenWindow(&ConsoleWindow)))
      {
      printf("Window will not open!\n");
      Close_All();
      exit(FALSE);
      }


   consoleWritePort = CreatePort("wgb-con-dev.write", 0L);

   if (!consoleWritePort)
      {
      printf("The new WritePort not enabled!\n");
      Close_All();
      exit(FALSE);
      }

   consoleWriteMsg  = CreateStdIO(consoleWritePort);
   if (!consoleWriteMsg)
      {
      printf("CreateStdIO for Write not enabled!\n");
      Close_All();
      exit(FALSE);
      }
```

```
   consoleReadPort = CreatePort("wgb-con-dev.read", 0L);
   if (!consoleReadPort)
      {
      printf("The new ReadPort not enabled!\n");
      Close_All();
      exit(FALSE);
      }

   consoleReadMsg  = CreateStdIO(consoleReadPort);
   if (!consoleReadMsg)
      {
      printf("CreateStdIO for Read not enabled!\n");
      Close_All();
      exit(FALSE);
      }


   consoleWriteMsg->io_Data   = (APTR)FirstWindow;
   consoleWriteMsg->io_Length = sizeof(*FirstWindow);

   conDevErr = OpenDevice ("console.device", 0L, consoleWriteMsg,
0L);
   if (conDevErr)
      {
      printf("OpenDevice error!\n");
      Close_All();
      exit(FALSE);
      };

   consoleReadMsg->io_Device  = consoleWriteMsg->io_Device;

   consoleReadMsg->io_Unit    = consoleWriteMsg->io_Unit;

   }


/*************************************
 *                                   *
 * Function: Close All               *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     20.10.1987                *
 *                                   *
 * No Parameter                      *
 *                                   *
 *************************************/


Close_All()

   {
   if (!conDevErr)
      CloseDevice(consoleWriteMsg);
```

```
   if (consoleWriteMsg)
      DeleteStdIO(consoleWriteMsg);

   if (consoleReadMsg)
      DeleteStdIO(consoleReadMsg);

   if (consoleWritePort)
      DeletePort(consoleWritePort);

   if (consoleReadPort)
      DeletePort(consoleReadPort);

   if (FirstWindow)
      CloseWindow(FirstWindow);

   if (IntuitionBase)
      CloseLibrary(IntuitionBase);
   }


/*************************************
 *                                   *
 * Function: Output with the ConsoleDev*
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     23.10.1987                *
 *                                   *
 * request  Message Port             *
 * Zchar    Number of the character  *
 *                                   *
 *************************************/


ConWriteChr(request, Zchar)

struct IOStdReq *request;
char            Zchar[];

   {
   ConWrite(request, Zchar, 1);
   }


  /*******************************
  ConWriteString(request, Text)

  struct IOStdReq *request;
  char            Text[];

     {
     ConWrite(request, Text, -1);
     }
  *******************************/
```

```
ConWrite(request, Text, Length)

struct IOStdReq *request;
char            Text[];
int             Length;

   {
   request->io_Command = CMD_WRITE;
   request->io_Data    = (APTR)Text;
   request->io_Length  = Length;
   DoIO(request);
   }


/*************************************
 *                                   *
 * Function: input with the ConsoleDev *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb     24.10.1987                *
 *                                   *
 * request  Message Port of report   *
 * whereto  Address of text buffers  *
 * Length   Length of  Text buffers  *
 *                                   *
 *************************************/


QueueRead(request, whereto, Length)

struct IOStdReq *request;
char            *whereto;
ULONG            Length;

   {
   request->io_Command = CMD_READ;
   request->io_Data    = (APTR)whereto;
   request->io_Length  = Length;
   SendIO(request);
   }


/*********************************************

int ConMayGetChr(consolePort, request, whereto)

struct Port     *consolePort;
struct IOStdReq *request;
char            *whereto;

   {
   register temp;
```

```
   if (GetMsg(consolePort) == NULL)
      return(FALSE);

   temp = *whereto;
   QueueRead(request, whereto, 1L);
   return(temp);
   }


UBYTE ConGetChr(consolePort, request, whereto)

struct IOStdReq *request;
struct Port     *consolePort;
char            *whereto;

   {
   register temp;
   while (GetMsg(consolePort) == NULL)
      WaitPort(consolePort);

   temp = *whereto;
   QueueRead(request, whereto, 1L);
   return(temp);
   }

***********************************************/


/*************************************
 *                                   *
 * Function: Text buffer positioning *
 * ================================= *
 *                                   *
 * Author: Date:       Comments:     *
 * ------  ----------  ----------    *
 * Wgb      11/2/1987                *
 *                                   *
 * y  Number of the line             *
 * x  Number of the column           *
 *                                   *
 *************************************/


Clear_Buffer()
   {
   int y;
   for(y=0; y<80; y ++)
      Clear_Line(y);
   }


Clear_Line(y)
int y;
   {
   int x;
   for(x=0; x<79; x ++)
```

```
      WindowBuffer[y][x] = 0x20;
   WindowBuffer[y][78] = NULL; /* End odf line marker */
   WindowBuffer[y][79] = NULL; /* Number of entered characters */
   }


Move_Up()
   {
   int x, y;
   for(y=1; y<80; y ++)
      for(x=0; x<80; x ++)
         WindowBuffer[y-1][x] = WindowBuffer[y][x];
   Clear_Line(79);
   }


Move_Down()
   {
   int x, y;
   for(y=79; y>0; y --)
      for(x=0; x<80; x ++)
         WindowBuffer[y][x] = WindowBuffer[y-1][x];
   Clear_Line(0);
   }
```

3.10 User-Defined Keyboard Tables

We have already said that the console.device converts the RAWKEY codes into characters. Now we'll look at which assignments, commands or definitions are allowed by this conversion.

A table helps translate the RAWKEY codes to the proper characters. The table indicates the original key and the character to which it should be converted.

This keyboard table also contains sections for keys pressed together with the <Shift>, <Alt> <Ctrl> and/or <CapsLock> keys. The <CapsLock> codes, for example, affect uppercase letters only, but not numbers or special characters.

Keyboard tables also allow you to assign text strings of normal or special characters to a single key.

3.10.1 Keyboard Table Design

We now know that the translation of the RAWKEY codes occurs through keyboard tables. We should next examine the structure of one of these tables, and which characters such a table should contain.

The term *keyboard table* refers to the large table, as well as the smaller subtables which comprise the large table.

The KeyMap structure states the contents of the subtables. It contains a pointer to each subtable.

```
struct KeyMap
{
0x00  00  UBYTE *km_LoKeyMapTypes;
0x04  04  ULONG *km_LoKeyMap;
0x08  08  UBYTE *km_LoCapsable;
0x0C  12  UBYTE *km_LoRepeatable;
0x10  16  UBYTE *km_HiKeyMapTypes;
0x14  20  ULONG *km_HiKeyMap;
0x18  24  UBYTE *km_HiCapsable;
0x1C  28  UBYTE *km_HiRepeatable;
0x20  32
};
```

You can see that the KeyMap structure actually has two separate sections—a Lo section and a Hi section.

The Lo section contains information about characters with RAWKEY codes from 0x00 to 0x3F. The Hi section contains information about characters with RAWKEY codes from 0x40 to 0x6. This division exists to keep the keyboard open for more keys which can be entered into the Hi section. This makes enhanced keyboards possible.

Examine the RAWKEY diagram below, with which you can get the code for any key:

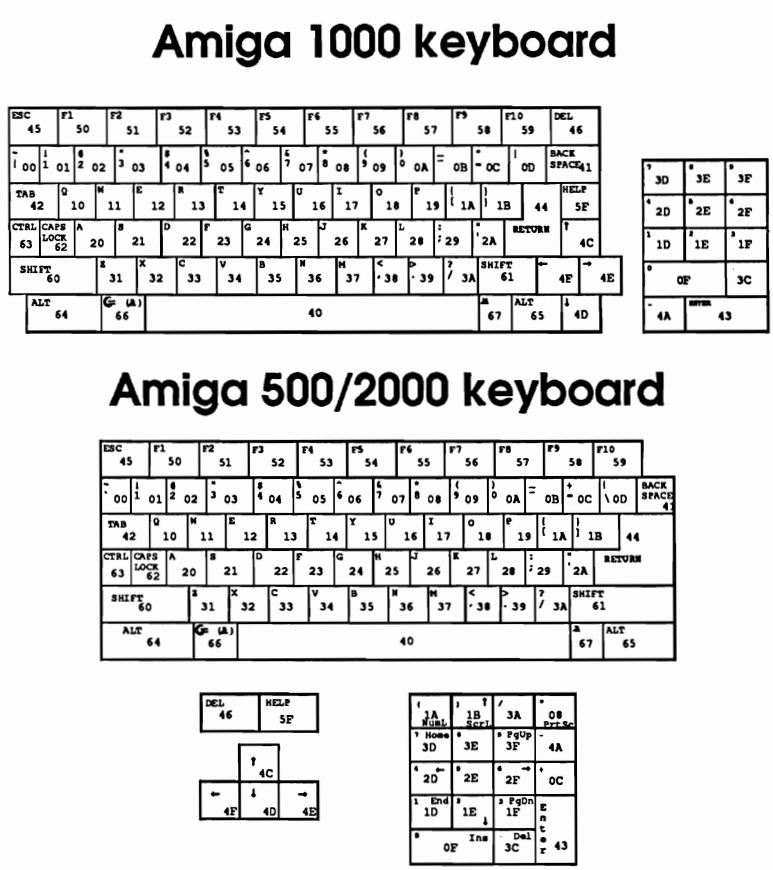


Figure 3.13

Let's look at each table:

```
km_LoKeyMapTypes
km_HiKeyMapTypes
```

These two tables contain one byte for every key. This byte lists the flag information described below. Unfortunately it's almost impossible to

list the combinations for every key. These combinations include normal keys, <Shift>ed keys, <Ctrl> combinations and <Alt> combinations. Because the key combinations from two qualifiers, the <Shift>, <Alt> and the <Ctrl> key, are considered, only four characters can be used per key. You can test to see which qualifier releases which characters. Or you could decide upon an entire string for the key, but then you should take into consideration that only two arrangements are allowable for the key.

```
km_LoKeyMap
km_HiKeyMap
```

The ULONG values of these tables contain the characters that should appear when a key is pressed. Since the ULONG value consists of four bytes, each byte has the character code belonging to one of the four combinations. This is because the tables handle the character as a string. Then the ULONG value represents a pointer to the string.

```
km_LoCapsable
km_HiCapsable
```

These two tables contain no displayable character codes. We find only set and unset bits for each RAWKEY code in these tables. A set bit indicates that the <CapsLock> key is active, i.e., this key should be handled like a <Shift>ed key. An unset bit displays the key in normal form.

```
km_LoRepeatable
km_HiRepeatable
```

These two tables work much like the Capsable tables. Here again we have a bit for each RAWKEY code. A set bit indicates key repetition according to values set by Preferences. An unset bit disables key repeat.

3.10.2 Working with the Keyboard Tables

When you want to take control of the keyboard tables, you must know the current status of the tables. This task involves the console.device. You can only work with the keyboard tables if the console.device is active.

There are commands that send information about the current status to the console.device. It is possible to insert the pointer to a new keyboard table. Unfortunately, the two operations don't execute through library controlled functions. The entries are transmitted entirely through IOStdReq. That is why we recommend that you write some of your own functions.

The CD_ASKKEYMAP function gets the current KeyMap arrangement. We need the size and starting address of our KeyMap structure, into which the data should be placed. Then we can call the request:

```
AskKeyMap(ConWriteRequest, OwnKeyMap)

struct IOStdReq *ConWriteRequest;
struct KeyMap   *OwnKeyMap;

{
   BYTE Error;
   ConWriteRequest->io_Command = CD_ASKKEYMAP;
   ConWriteRequest->io_Lenght  = sizeof(struct KeyMap);
   ConWriteRequest->io_Data    = &OwnKeyMap;

   DoIO(ConWriteRequest);

   Error = ConWriteRequest->io_Error;

   if (Error)
      return(FALSE);
   else
      return(TRUE);
}
```

The pointer to the active keyboard tables appears in the OwnKeyMap structure after executing the above function. This can be examined and changed. With these methods you can, for example, change a single character on the keyboard.

This method is enough for one or two characters, but you need something more powerful when you have a very unusual keyboard arrangement that cannot be activated with a few changes, or when you want to load an entire table. Here's the alternative process. First the keyboard values are stored in memory, either within the program text or by loading. Then the program creates the KeyMap structure. This structure is the primary point from which the console.device gets its information. Then the new KeyMap must be transferred to the console.device:

```
SetKeyMap(ConWriteRequest, OwnKeyMap)

struct IOStdReq *ConWriteRequest;
struct KeyMap   *OwnKeyMap;

{
   BYTE Error;
   ConWriteRequest->io_Command = CD_SETKEYMAP;
   ConWriteRequest->io_Length  = sizeof(struct KeyMap);
   ConWriteRequest->io_Data    = &OwnKeyMap;

   DoIO(ConWriteRequest);

   Error = ConWriteRequest->io_Error;
```

```
   if (Error)
      return(FALSE);
   else
      return(TRUE);
}
```

This is almost the same function as listed previously. The single difference is the different function call. If you want to make one function perform both tasks, all you need is one parameter to extend the function call:

```
DoKeyMap(WriteRequest, KeyMapPointer, CD_SETKEYMAP)

DoKeyMap(ConWriteRequest, OwnKeyMap, Mode)

struct IOStdReq *ConWriteRequest;
struct KeyMap   *OwnKeyMap;
UWORD            Mode;

{
   BYTE Error;
   ConWriteRequest->io_Command = Mode;
   ConWriteRequest->io_Length  = sizeof(struct KeyMap);
   ConWriteRequest->io_Data    = &OwnKeyMap;

   DoIO(ConWriteRequest);

   Error = ConWriteRequest->io_Error;

   if (Error)
      return(FALSE);
   else
      return(TRUE);
}
```

This last function lets you address two more functions named CD_ASKDEFAULTKEYMAP and CD_SETDEFAULTKEYMAP. Both work exactly as the names suggest. These functions let us control the default KeyMap (the KeyMap made available when the console.device opens).

This KeyMap contains all of the characters divided into the groups Lo and Hi. The Lo group is handled as follows:

- Any keys pressed by themselves (i.e., no <Shift>, <Ctrl> or <Alt>) create their normal ASCII codes.
- Any keys pressed with the <Shift> key create their shifted ASCII codes.
- Any keys pressed with the <Alt> key create their ASCII codes with the highest bit set (0x80).

- Any keys pressed with the <Alt> and <Shift> keys create their shifted ASCII codes with the highest bit set (0x80).
- Any keys pressed with the <Ctrl> key create their ASCII codes with bits 5 and 6 deleted.

The Hi group contains only control codes that don't consist of individual codes: We find only strings here. That is also the reason why the control keys can be placed two in a group at most.

Keys without any qualifier

Key	Translated value
<Backspace>	0x08
<Enter>	0x0D
<Del>	0x7F

Keys with one qualifier

Key	without	with	qualifier
<Space>	0x20	0xA0	Alt
<Return>	0x0D	0x0A	Ctrl
<Esc>	0x1B	0x9B	Alt
-	0x2D	0xFF	Alt

Keys of High-KeyMap

Key	Value	Value with Shift
<Tab>	0x09	0x9B "Z"
Up	0x9B "A"	0x9B "T"
Down	0x9B "B"	0x9B "S"
Forward	0x9B "C"	0x9B " @"
Backward	0x9B "D"	0x9B " A"
F1	0x9B "0~ "	0x9B "10~ "
F2	0x9B "1~ "	0x9B "11~ "
F3	0x9B "2~ "	0x9B "12~ "
F4	0x9B "3~ "	0x9B "13~ "
F5	0x9B "4~ "	0x9B "14~ "
F6	0x9B "5~ "	0x9B "15~ "
F7	0x9B "6~ "	0x9B "16~ "
F8	0x9B "7~ "	0x9B "17~ "
F9	0x9B "8~ "	0x9B "18~ "
F10	0x9B "9~ "	0x9B "19~ "
HELP	0x9B "?~ "	

3.10.3 Creating Your Own Keyboard Table

You already know that the keyboard table is actually a group of smaller tables. We'll now explain how you can create your own table, and what to watch out for when making it.

KeyMap Types

We first need the two table types. These contain, as explained above, a check to see if it is a string or four single characters, and the qualifiers which access every key. The following flags are possible:

KeyMap Types:

Name	Value	Description
KC_NOQUAL	0L	No qualifier
KC_VANILLA	7L	Standard value with <Ctrl>
KCF_SHIFT	0x01L	<Shift> key pressed
KCF_ALT	0x02L	<Alt> key pressed
KCF_CONTROL	0x04L	<Ctrl> key pressed
KCF_DOWNUP	0x08L	Reacts when key is released
KCF_DEAD	0x20L	No key reaction
KCF_STRING	0x40L	Key displays a string

We can only reach different arrangements by choosing the type. The four bytes have the following meanings:

KeyMap Type combinations:

Qualifier combination	Byte 1	Byte 2	Byte 3	Byte 4
KC_DEAD	-	-	-	-
KC_NOQUAL	-	-	-	without
KC_SHIFT	-	-	Shift	without
KC_ALT	-	-	Alt	without
KC_CONTROL	-	-	Ctrl	without
KC_ALT + KC_SHIFT	Shift+Alt	Alt	Shift	without
KC_CONTROL + KC_ALT	Ctrl+Alt	Ctrl	Alt	without
KC_CONTROL + KC_SHIFT	Ctrl+Shift	Ctrl	Shift	without
KC_VANILLA	Shift+Alt (Ctrl also accessible)	Alt	Shift	without
KC_STRING	Pointer to string			

The option of having any table arrangement should be available in your programs. The most frequently used flag is KC_VANILLA because the <Shift> and <Alt> keys are integrated. <Ctrl> is also supported, with bits 5 and 6 of the NOQUAL value unset.

KeyMap A Lo or Hi KeyMap can be easily put together. C programmers should do this using assembler, because the problem is easier to solve there. You can easily complete a table divided into four groups. Each group has a key equivalent assigned according to the above table. Every byte contains a character that is displayed according to the key combination.

When the key assignment is handled as a string, the system replaces the group of four bytes with a LONG value that represents the pointer to our string. Unfortunately, not every string is displayed the same. The pointer first points to another table that contains the value for this key. The string length appears first, which should appear with a normal key press. Next an offset appears pointing from the beginning of the string table to the string itself.

We find the string length listed as another value that should be given as a <Shift>ed key press. The offset appears at the beginning of the table.

Capsable Like a typewriter, the Amiga keyboard also has a <CapsLock> key that acts as a continuous <Shift>. Because the keyboard has a keyboard processor, it isn't just a matter of simulating the continuous <Shift>. The processor executes its own form of <Shift>. The <CapsLock> key tests to see which keys are usable and which are not. This method has the advantage that punctuation remains unshifted, and can be entered as normal without having to release the <CapsLock> key.

We have 8 bytes with 64 bits for all of the keys. Each of these bits represents a key:

```
Bit 0 Byte 0 RAWCODE 0x00, Bit 1 Byte 0 RAWCODE 0x01 ...
Bit 0 Byte 0 RAWCODE 0x08 ... Bit 7 Byte 7 RAWCODE 0x4f.
```

Set all of the bits for those keys which you want <Shift>ed.

Repeatable The same method available for the Capsable table is used for the Repeatable table. Each set bit indicates that the key should be repeated according to the repetition time and frequency set in Preferences.

Many keys can be excluded from the repeat function. That's important if a key doesn't have to repeat, or if repeating could cause trouble. Usually the <Return> key does not repeat.

3.11 Memory Management

Some programs may require access to a certain area of memory. The simplest memory forms are the variables. Here you control what is needed for program formation. Arrays are based upon these variable forms. But you may have heard quite a bit about how limiting these can be.

Here's where the problem arises. The program could need a variable array double the size of the number expected for program execution. This could have a dimension of 1000*1000 for the color values of a high resolution graphic. Our compiler reserves a doubled variable in the finished program 100000 times. You can imagine how much memory that requires.

We would like to show you a short example which accesses memory areas from the system and frees them after use. Three methods of memory organization are demonstrated. The example itself is chosen from Intuition. We will try to assign four text buffers for the corresponding string gadgets.

3.11.1 Memory Organization of the Amiga

Before we can write a program, you should have a working knowledge of the Amiga's memory layout. The Amiga has a twofold dynamic memory management.

This table lists the memory arrangement:

Address	Arrangement	Comments
0x00000000	Chip RAM	Copy of the Kickstart ROM
0x08000000		Third copy of Chip RAM
0x20000000	8 MByte	Fast RAM
0xA0000000	CIAs	
0xC0000000	512K expansion	(Amiga 500 & 2000 only)
0xC8000000	Unused	
0xDC000000	Real time clock	(Amiga 500 & 2000 only)
0xDF000000	Custom chips	
0xE0000000	Unused	
0xE8000000	Expansion slots	
0xF0000000	ROM module	
0xF8000000	Kickstart ROM copy	
0xFC000000	Kickstart ROM	

The entire memory is divided into two regions. The first 512K is chip RAM, which means it can be addressed from the custom chips. The second region consists of fast RAM or ROM, and can be addressed from the 68000.

The word "dynamic" comes from the fact that the 8MB fast RAM doesn't exists in the basic Amiga: Memory can be expanded dynamically.

When we want our program to request an area of memory from the system, then we must specify whether it should be chip RAM or fast RAM. If we give no specification, fast RAM is used first. If this doesn't succeed, the memory is is taken from chip RAM.

Note: Some older programs may cause incompatibility problems with memory expansion. Without memory expansion you are left with only chip RAM. This memory can also be addressed from the custom chips. Adding memory expansion adds fast RAM to the system. The system may crash when older application programs attempt access to the memory in this region.

3.11.2 First Steps with `AllocMem()`

Our first try at memory control uses an `Exec` function named `AllocMem()`. This function allocates a memory area according to specific size and criteria. You assign two arguments and receive the address of the added memory in return. If no memory can be found with the specified size and attributes, the system returns a null pointer (error).

```
MemoryBlock = AllocMem(ByteSize, Requirements);
    D0          -198      D0         D1
```

The flags `MEMF_FAST` and `MEMF_CHIP` describe the attributes of the memory area. `MEMF_PUBLIC` indicates memory usable for different tasks. The `MEMF_PUBLIC` flag is currently unimplemented. However, you should use it anyway to maintain compatibility with later Amiga systems. `MEMF_CLEAR` clears the area of memory (i.e., fills the area with null bytes)., which means it should be filled with zeros. The memory area must be a minimum of eight bytes long.

The following program reserves memory for four text buffers:

```
/*************************************
 *                                   *
 * Memory Management  : AllocateMemory *
 * ================================= *
 *                                   *
```

```
 * Author:  Date:      comments:       *
 * ------  ----------  ----------      *
 * Wgb     10/16/1987  only for test   *
 *                     purposes        *
 *                                     *
 ***************************************/


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

#define MemoryType MEMF_CHIP | MEMF_CLEAR


UBYTE *UndoBuffer, *FileBuffer, *DiskBuffer,
                   *SuffBuffer, *AllocMem();


main()
   {
   UndoBuffer = AllocMem(512L, MemoryType);
   if (!UndoBuffer)
      {
      printf("Problems with the Undo buffer!\n");
      FreeMemory();
      exit(FALSE);
      }

   FileBuffer = AllocMem(30L, MemoryType);
   if (!FileBuffer)
      {
      printf("Problems with the FileBuffer!\n");
      FreeMemory();
      exit(FALSE);
      }

   DiskBuffer = AllocMem(512L, MemoryType);
   if (!DiskBuffer)
      {
      printf("Problems with the DiskBuffer!\n");
      FreeMemory();
      exit(FALSE);
      }

   SuffBuffer = AllocMem(10L, MemoryType);
   if (!SuffBuffer)
      {
      printf("Problems with the SuffBuffer!\n");
      FreeMemory();
      exit(FALSE);
      }


   printf("Memory reserved!\n");

   FreeMemory();
```

```
    exit(TRUE);
    }


FreeMemory()
    {
    if (UndoBuffer)  FreeMem(UndoBuffer, 512L);
    if (FileBuffer)  FreeMem(FileBuffer, 30L);
    if (DiskBuffer)  FreeMem(DiskBuffer, 512L);
    if (SuffBuffer)  FreeMem(SuffBuffer, 10L);

    printf("Memory free again!\n");
    }
```

Program Description

The include file <exec/memory.h> contains the memory type definition. We link the two files together and then define a label. Then we need the four pointers to the memory areas. The function is supplied with a value for the required memory.

The main program tries to allocate the memory areas. If this doesn't work it branches to the free routine and the program stops. The free routine works similar to the Close_All() function. Every pointer is checked to see if it points to a memory block. After this is done the memory will be freed. This avoids the error that occurs when memory is freed that was not allocated.

To release the memory we use another Exec function:

```
FreeMem(MemoryBlock, ByteSize);
 -210       A1          D0
```

The disadvantage of this method is that we must free every memory area, and know the memory size. That is unnecessary work for the programmer in both computation time and program size.

3.11.3 Improvement using AllocEntry()

Often multiple memory areas have different needs. The AllocMem() function is usable for this, but not a very good solution. A structure would be helpful. This structure contains all of the memory areas with their features. We then assign the pointer to AllocEntry() and it attempts to allocate all of the memory areas.

The basic structure is named MemList:

```
struct MemList
{
0x00  00  struct Node ml_Node;
0x0E  14  UWORD ml_NumEntries;
```

```
0x10  16  struct MemEntry ml_ME[1];
0x18  24
};
```

The old node is needed at the beginning, as usual. The `ml_NumEntries` function states the number of entries on the memory list. The second `MemEntry` structure gives additional information to the first memory block.

```
struct MemEntry
{
union
   {
   ULONG meu_Reqs;
   APTR meu_Addr;
   }
0x00  me_Un;
0x04  04  ULONG me_Length;
0x08  08
};
```

Definition:

```
me_un me_Un
me_Reqs me_Un.meu_Reqs
me_Addr me_Un.meu_Addr
```

The value `me_Un` contains the definition of the features of this memory block at the beginning. The starting address appears here according to the allotment. The length of the memory area is reserved in `me_Length`.

From these two structures, `MemList` and `MemEntry`, we must assemble a structure before we can use `AllocEntry()`. This can look something like this:

```
struct Memoryneeds
   {
   struct MemList Head;
   struct MemEntry UndoBuffer;
   struct MemEntry FileBuffer;
   struct MemEntry DiskBuffer;
   struct MemEntry SuffBuffer;
   } Memory;
```

This self-contained structure is filled with the desired values. Then `AllocEntry()` can be called. It tries to get the memory, and we find the data in the individual `MemEntry` structures.

```
MemList = AllocEntry(Entry);
   D0       -222      A0

          FreeEntry(Entry);
            -228     A0
```

Here is the complete listing:

```
/*************************************
 *                                   *
 * Memory management  : MemoryEntries *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     10/16/1987  only for test *
 *                     purposes      *
 *                                   *
 *************************************/


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

#define MemoryType MEMF_FAST | MEMF_CLEAR


struct MemList *MemoryPtr, *AllocEntry();
APTR UndoData, FileData, DiskData, SuffData;

struct Memoryneeds
   {
   struct MemList Head;
   struct MemEntry UndoBuffer;
   struct MemEntry FileBuffer;
   struct MemEntry DiskBuffer;
   struct MemEntry SuffBuffer;
   } Memory;


main()
   {
   Memory.Head.ml_NumEntries   = 5;

   Memory.Head.ml_me[0].me_Length = 0;
   Memory.UndoBuffer.me_Reqs   = MemoryType;
   Memory.UndoBuffer.me_Length = 512L;
   Memory.FileBuffer.me_Reqs   = MemoryType;
   Memory.FileBuffer.me_Length = 30L;
   Memory.DiskBuffer.me_Reqs   = MemoryType;
   Memory.DiskBuffer.me_Length = 512L;
   Memory.SuffBuffer.me_Reqs   = MemoryType;
   Memory.SuffBuffer.me_Length = 10L;

   MemoryPtr = (struct MemList *)AllocEntry(&Memory);

   if ((ULONG)MemoryPtr & (1<<31))
      {
      printf("The list cannot be posted!\n");
      exit(FALSE);
      }
   else
```

```
    {
    UndoData = MemoryPtr->ml_me[1].me_Addr;
    FileData = MemoryPtr->ml_me[2].me_Addr;
    DiskData = MemoryPtr->ml_me[3].me_Addr;
    SuffData = MemoryPtr->ml_me[4].me_Addr;
    }

  printf("Memory allocated!\n");

  FreeEntry(MemoryPtr);
  printf("Memory is free again!\n");
  exit(TRUE);
  }
```

3.11.4 The Final Solution

The two preceding functions are from the `Exec` library. Since this chapter is about C and Intuition, we've saved the memory management functions supported by Intuition until now.

The two functions are called `AllocRemember()` and `FreeRemember()`. As the names suggest, Intuition helps us by marking the reserved areas. We assign the function the size of the desired memory area, the memory type and the pointer to a `Remember` structure. Intuition either returns an error message or the address of the memory area. The `Remember` structure is specifically for memory management under Intuition, and "remembers" the used memory areas. If you want to release the memory area, assign `FreeRemember()` the pointer to the `Remember` structure. This releases all the previously reserved blocks.

Here are the two functions with their general syntax:

```
MemBlock = AllocRemember(RememberKey, Size, Flags);
    D0          -396          A0         D0     D1

           FreeRemember(RememberKey, ReallyForget);
               -408          A0            D0
```

The `Remember` structure looks like this:

```
struct Remember
{
0x00  00  struct Remember *NextRemember;
0x04  04  ULONG RememberSize;
0x08  08  UBYTE *Memory;
0x0C  12
};
```

Structure description

The beginning of the structure is a pointer to another `Remember` structure. Multiple memory blocks are thus linked to a list. `RememberSize` tells the size of the memory, while `*Memory` represents the address.

The trick is that you first provide a pointer to a `Remember` structure that is not yet present, called `RememberKey`, and this is set to zero. This pointer is passed to `AllocRemember()`, and then the first `Remember` structure becomes active. The `Remember` list gets larger with every call to `RememberKey`.

`FreeRemember()`'s task is very simple. The function goes back and forth in the list and releases every area of memory. The memory that the list itself occupies can be released if the programmer wishes.

The advantage to this system lies in the easy release of all of the different areas of memory, and in the organization maintained by Intuition. Here's the program:

```
/*************************************
 *                                   *
 * Memory Management  : Remember     *
 * ================================= *
 *                                   *
 * Author:  Date:      Comments:     *
 * ------  ----------  ----------    *
 * Wgb     16.10.1987  only for test *
 *                     purposes      *
 *                                   *
 *************************************/


#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>

#define MemoryType MEMF_CHIP | MEMF_CLEAR


struct IntuitionBase *IntuitionBase;
struct Remember      *RememberPtr = NULL;

VOID *OpenLibrary();

UBYTE *UndoBuffer, *FileBuffer, *DiskBuffer,
                   *SuffBuffer, *AllocRemember();


main()
   {
   if (!(IntuitionBase = (struct IntuitionBase *)
         OpenLibrary("intuition.library", 0L)))
      {
      printf("No Intuition Library found!\n");
```

```
      exit(FALSE);
      }

   UndoBuffer = AllocRemember(&RememberPtr, 512L,
MemoryType);
   FileBuffer = AllocRemember(&RememberPtr,  30L,
MemoryType);
   DiskBuffer = AllocRemember(&RememberPtr, 512L,
MemoryType);
   SuffBuffer = AllocRemember(&RememberPtr,  10L,
MemoryType);

   printf("Memory reserved!\n");

   FreeRemember(RememberPtr, TRUE);
   printf("Memory free again!\n");
   CloseLibrary(IntuitionBase);
   }
```

4. Operating System Programming

4. Operating System Programming

This chapter contains a large, real-world application written in C. We've included this listing for two reasons:

1.) You'll be able to use this editor, which is much more flexible than ED for entering your own C source codes;

2.) This listing demonstrates how to combine different components of the operating system.

We selected a text editor as a demonstration program because it uses a relatively large number of the operating systems functions. You don't have to worry about programming the operating system; your programming task just involves memory management, text output and keyboard input. As we said above, you can use this editor to write your own C programs. Later in the chapter we'll show you how to combine assembler code with C source code, and how to search for errors using the debugger.

In addition to the normal capabilities that any editor has, our editor includes a few extra features for creating program source code in general. This means that you can use the editor for writing source code in most programming languages, not just in C.

The editor was compiled with the Aztec C compiler. However, the source code is flexible enough that it should compile with any compiler. When you find special compiler functions or utility program calls, the editor code is written so that other compilers can use this.

4.1 Planning the Editor

When you begin a new program, you should first thoroughly determine what the program should do. Write down the task you want the final product to perform, then break it down into the segments needed to perform this task. This helps you see logical errors before programming, and helps eliminate large programming changes during the development stages. You should write down all the functions needed, problems that must be solved, and possible solutions for these problems. This list will be used later in the development stage. We'll

start this brainstorming session with a list of four requirements of our editor:

1.) fast;

2.) flexible;

3.) programmable;

4.) user-friendly.

Fast

Let's begin with the first requirement: The editor should be fast. Placing the source text in memory plays a major role in speed. The more organized the memory, the faster text can be manipulated. You can allocate editing memory with a given size, and load the text into that memory range as a series of blocks. The advantage to this is a minimum of memory management. In addition, source files load and save quickly.

Buffers

Problems arise when you try inserting characters in the text. If you insert a character in a large text, the editor becomes unbearably slow because a relatively large memory area must be moved. It would be better if you could copy the line containing the entry into a buffer and insert the character there. Keeping this buffer small speeds up the memory movement. The buffer is then copied into the text if the user moves the cursor out of the line. This method works well enough if you have spare memory available and want to edit a short text. However, it can take a long time to shift the edited text, especially for files longer than 30K.

There is also a disadvantage during scrolling. To move from one line to the next you must search for the end of the line. This takes somewhat longer than computing the beginning of the next line.

Arrays offer another option for placing text in memory. You can determine the maximum number of characters per line and the maximum number of lines, then define an array with the corresponding dimensions: `linefield[maxline][maxcolumn]`. That way you have fast access to individual lines and characters within a line. Characters can be inserted directly in the text. This method of insertion is fast because you only move the characters of the current line. This method also has disadvantages:

- If you want to insert new lines, you must move the entire text.
- The array size is preset. If you want a larger array, you must first change the size of the array, thereby deleting its contents.
- Not all of the lines use the maximum line length. Trial and error has shown that only a few lines reach maximum length, while most lines only use a third to one half of the allowable line width.

Linked lists

Because of these disadvantages you can choose another method and store the lines as linked lists. Then available memory limits the maximum number of lines instead of an array, and changing the pointer moves you to the next line. The lines have no set length so short lines require little memory. The resulting memory management is complicated as you'll see in the development stage, but the bottom line is a good compromise between speed and memory.

After you know how your lines are constructed, you must consider from where you'll get the memory for each line. The operating system can supply the memory at any time. You can utilize this memory by dividing blocks of memory into smaller units (the operating system cannot produce these smaller units). It is better if you allocate a block of a predetermined size (e.g., 5K) from the operating system and divide this up yourself, according to the present need. Write the functions so you don't need to think about the "why" later when you want to add a new line to the program. We'll go into more detail when we discuss memory management.

User-friendly

Now for a few ideas to make the program more user-friendly. An `Undo` function will remove the changes just made to a line. To create the `Undo` function you must store the original text in a buffer, delete the changed line and restore the line's original contents from the buffer.

A feature that has always bothered us when working with editors and word processors is when the program cannot sense text changes. When leaving an editor it usually asks, "Are you sure that you have saved the text?" whether or not changes have been made since the last save. The editor would be a friendlier program if it only asked this question after the text has been changed since the last save. To determine whether the text was changed, a flag must be set whenever a change is made. We mention this now because if you forget to set the flag in a few places in the code, the program won't work. The error will be hard to find if you don't know about this flag.

The editor should support multiple windows. This is important when doing a lot of work if you want to move blocks of text between different source codes. When you only have one window, this results in a lengthy load and save. But you can't just open more windows. Each window requires its own editing memory, its own cursor and other things. This means that you cannot globally save all of the important information about the text, like number of lines, cursor position, etc. Instead it must be saved to a structure for which exactly one structure exists for each editor window. There is still a pointer to the actual structure, whose window is still active. This means more work for you at the beginning of the development stage, but once you've implemented a second window, providing for the others becomes easier.

An interesting capability of the Amiga is the option of assigning a character string to any key. You can, for example, specify that the Amiga sends a "ae" or "\344" (octal code for a) every time you press

<Alt><A>. Or you can specify whether the <Return> key sends a linefeed (LF), a carriage return (CR) or a combination of the two (CRLF). You can also assign special characters to any key. The possibilities given here will be described later in this chapter.

Flexibility This program should be flexible. That is, instead of the programmer laying out certain unchangeable parameters in the source code, the user should be able to change some parameters to suit his own needs.

Tabs Your editor should be able to insert true tabs instead of just blank spaces. Structured programming involves inserting spaces to indent lines, which wastes memory. If you own an Aztec C compiler, you could use the Z editor which inserts true tabs. However, Z is a very difficult editor to use and learn. It would be better if you could set the tabs anywhere you want. This is difficult to program, but this tab system makes the editor more flexible.

Auto indent Automatic indentation could be indirectly tied in with tabs. When you press <Return> at the end of a line, the cursor should move to the same first column as the previous line (i.e., the first character that is neither a blank nor a tab). This auto indent saves the user's inserting spaces or tabs at the beginning of each line for structured programming.

The editor should also display special and control characters. When ED encounters unusual characters it displays the message "file contains binary" and stops. Control characters use the lower half of the ASCII character codes. Because these characters aren't displayable on the Amiga, you could display these in a different color and add a constant value to the ASCII code of the character. This then returns a visible character. The character with ASCII code 1 appears as a red "A" in the text (Code("A") = 65 = 1 + 64). You could display the reserved C keywords in bold print. This would make the program easier to read.

Folding The final user-friendly feature we could include in the editor is folding. Folding lets you make a section of program code invisible without actually deleting the section. For example, if you have two program lines that you wish to view, but the two lines are separated by other program codes, you can fold the unnecessary text away.

Programmable We now turn to the programmability of the editor. All of the editor's functions should be accessible from simple commands. You can enter multiple commands in one line by placing semicolons between each command sequence. In addition, command sequences can be repeated by entering a number before a sequence (either single commands or multiple commands). So far this editor seems to do everything that ED does. We're going to take this editor a few steps further.

First we allow the use of variables which can provide numbers for cursor X/Y coordinates, strings or command sequences. If you place a numeric variable before a command, this command executes the number of times given in the variable. In addition you can have expanded

control structures, FOR-TO-DO, WHILE-DO and IF-THEN-ELSE constructs. Command sequences can be assigned to any function key. More ideas for commands could be implemented later, such as searching and replacing blocks of text (from line to line).

It would also be convenient to have access to CLI commands while in the editor. Then, for example, you would no longer need to save your text, exit the editor and call the compiler.

Data structures Once you have the complete list of ideas you must now think about the data structures which make up the editor. Data structures correspond to the important things which you must think about before you begin programming. All of the Amiga's data structures accessed by the operating system are of similar design; you only think about the lists. This makes manipulation of objects easier. The following line is the first data structure in your editor:

```
struct Zline
{
   struct Zline *succ;        pointer to the next line
   struct Zline *pred;        pointer to previous line
   UBYTE flags;               Diverse Flags
   UBYTE len;                 length of the line
   /* UBYTE line[] */
}
```

The two pointers later construct the linked list. You need two pointers because you want to move to either the previous line or the next line. The Amiga also has functions for doubled linked lists, which you can also use for your lines. If (`flags & 128`) is unequal to zero, the line appears in the buffer because the user changed this. This flag is needed because you must display the buffer if you move to a line where this flag is set. We will define more flags as the need arises.

Line length The `len` variable specifies the line length including the CR, LF or CR/LF that indicates the end of the line. By this method you can have a line without an end of line character if the line is longer than the screen's width. Instead of this line continuing off to the right of the screen, the line "wraps" and text appears on the next screen line without being split by a CR, LF or CR/LF. Word processors handle lines in this way.

The line itself directly follows `len`. Because of the changing lengths, you cannot include it in the structure. The length of the structure including the line appears as follows:

```
EntireLength = (sizeof(struct line) + line.len + 1) & -2.
```

`EntireLength` must be an even number, because the pointer in the Amiga must point to an even address in memory. This is a rule when working with the 68000 microprocessor. If you disobey this rule, you'll get a Guru Meditation number 00000003 (address error).

Memory blocks The next structure configures the memory blocks containing the text lines. A linked list connects the blocks of memory. The code needs a pointer to a list of all the available memory segments in this block, the length of the block, and the number of unused bytes. The structure looks like the following:

```
struct Memoryblock
{
   struct Memoryblock *succ;          Next memory block
   struct Memoryblock *pred;      Previous memory block
   ULONG length;             Maximum number of free bytes
   ULONG free;             Momentary number of free bytes
   struct FList freelist;            List of free pieces
};
```

The first two variables insert the memory blocks in a double linked list. The `length` variable gives the block size without the memory block structure. This is the equivalent of the number of free bytes present in an unused block (the length of a `memorysection` structure). `Free` states how many bytes are free in this block. Now the problem arises of where these free bytes are. These blocks can exist in any location, and are rarely in sequence. When a line is deleted, a free section of memory must not lie on another piece of free memory. Because every line is at least 10 bytes long (`sizeof(struct line)`!), you have 10 bytes in which you can construct a list of all of the free memory pieces in a block. Each element of this list has the following construction:

```
struct Memorysection
{
     struct Memorysection *succ;
     struct Memorysection *pred;
     UWORD len;
};
```

Management The first two elements are chosen with the operating system list in mind. They are handled as a double linked list for a set of management functions in the `Exec` library. Because you use the same chain, you can also use these functions for your lists without having to write your own. The element `len` contains the length of the entire memory section, including the `Memorysection` structure.

Next we need an `FList` structure, which has the same design as the list header of an operating system list:

```
struct FList
{
     struct Memorysection *head;
     struct Memorysection *tail;
     struct Memorysection *tailpred;
};
```

The structure should be familiar to you from the `Exec` lists (see the Appendix for an explanation of `Exec.library`). It is often said that

`head` points to the first element of the list, `tailpred` to the last element of the list, and `tail` is equal zero. This arrangement simplifies insertion and deletion in the lists.

This may sound like a lot of effort just to manage a line display. We could specify the memory for each line from the operating system, but this tends not to work too well. Remember that a text can consist of 1000 or more lines. Every time you change a line, you must release the old memory and allocate new memory. Once you've spent some time editing one text, a lot of gaps remain in memory.

When you want a new line, the program should first search for a memory section that has exactly the desired length. If it cannot find such a piece, take the memory from the largest memory section available. Using the largest section as a pattern, be sure that no smaller memory sections are reduced without needing reduction. For example, you need 14 bytes and the next largest memory piece has 16 bytes. If you distribute this, you get two memory pieces, one 14 bytes long and the other 2 bytes. These two bytes are leftovers. They can't be placed in the list of free memory pieces because you need at least 10 bytes for the `Memorysection` structure.

When you have no more room, allocate a new memory block and insert it in the block list.

That brings us to the last structure, the `Editor` structure. One exists for every editor window, and extracts all the important data needed for the window:

```
struct Editor
{
   struct Editor *succ;
   struct Editor *pred;
   struct Window *window;     Pointer to the editor window
   struct BList block;           List of the memory blocks
   struct ZList zlines;                  List of the lines
   UBYTE buffer[MAXWIDTH];                 Buffer to edit
   UBYTE tabstring[MAXWIDTH];                  Tabulators
   UWORD num_lines;                       Number of lines
   struct Zline *actual;       Pointer to the actual line
   struct Zline *top;                  Pointer to top line,
                          which is visible in the screen
   UWORD toppos;                     Number of the top line
   UWORD xpos,ypos;                          Cursor-Position
   UWORD changed:1;    It is 1 in case the text is changed
   UWORD insert:1;           It is 1 incase of Insert mode
};
```

Two more structures join the `Editor` structure, which represent the list heads:

```
struct BList
{
   struct Memoryblock *head;
   struct Memoryblock *tail;
   struct Memoryblock *tailpred;
};

struct ZList
{
   struct Zline *head;
   struct Zline *tail;
   struct Zline *tailpred;
};
```

The individual elements of the `Editor` structure are easy to prepare.

4.2 Development Stages

We want to begin developing the modules from which we can create the editor. The following source code opens the libraries which let the Amiga access operating system functions. The Amiga displays a message if the access is successful. Remember that you can only call functions of a library if you have opened it with `OpenLibrary`. It is also good practice to close all opened libraries.

```
/* <Editor.c> Version 0.0 */
/************
 *
 * Includes:
 *
 ************/

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

/***********
 *
 * Defines:
 *
 ***********/

#define REV 33L

/**********************
 *
 * External Functions:
 *
 **********************/

struct Library *OpenLibrary();
void CloseLibrary();

/*********************
 *
 * Global Variables:
 *
 *********************/

struct Library *IntuitionBase = NULL, *GfxBase = NULL;
struct Library *DosBase = NULL;

/*****************
 *
 * Main program:
 *
 *****************/
```

```
main()
{
   if ( !(IntuitionBase =
OpenLibrary("intuition.library",REV)))
      goto Ende;

   if ( !(GfxBase = OpenLibrary("graphics.library",REV)))
      goto Ende;

   if ( !(DosBase = OpenLibrary("dos.library",REV)))
      goto Ende;

printf("Everything is open\n");

Ende:
   if (DosBase) CloseLibrary(DosBase);
   if (GfxBase) CloseLibrary(GfxBase);
   if (IntuitionBase) CloseLibrary(IntuitionBase);
}
```

The `#include` functions add the data structure definitions needed to access the operating system routines. If you omit these commands, the C compiler displays error messages because of unknown structures. You should declare all functions that you use within a program at the beginning of the program. This is so the C compiler can determine whether you have specified the value of a function and a variable of the correct type.

Delays

Compiling takes some time, since the compiler needs to read the include files. This long delay also bothered the developers of the Aztec C compiler, so they added an option which precompiles include files. This precompilation links the files faster than with normal includes. For this reason, write a prelist file that contains only the `#include` instructions of your source text. The `editor.prelist` file located in the `pre` directory, looks like the following in our example:

```
<pre/editor.prelist>

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

Make a directory named PRE and save this file into it. To compile this file you must inform the compiler that it should create a file using the precompiled includes. The Aztec C compiler uses the `+H` option to declare this:

```
cc +Hpre/editor.pre pre/editor.prelist
```

This compiler command compiles the file containing the include instructions with the name "`editor.prelist`" in the `pre` directory. The file "`editor.pre`" is created in the same directory.

This file can be combined with the +I option when compiling the program:

```
cc +Ipre/Editor.pre src/Editor.c
```

The src (source) directory should contain the file Editor.c. The compiler should now work faster. The make utility program offers another possibility for simplifying the process. The make utility is used when a program consists of multiple modules that are linked together. It is better to divide large programs into smaller program modules. These modules are then compiled and put together with the linker. When modules are changed, the programmer must only compile the edited modules instead of the entire program.

Compiler and linker module control is handled by the make utility. You enter make, which reads the makefile. The makefile contains the modules your program consists of, and how these should be compiled and linked. If you call make, it checks the system date and time for any changes to sections of your program. It then automatically recompiles edited modules and links the compiled modules into a finished program.

You use the makefile to tell the make utility which files of your program depend on which other files. The object file of one module depends on the source text, and the finished program depends on the object files of all of the modules. Because of this dependence and the date and time when the file was created, make tests for changes and takes the appropriate steps to create the program. It is therefore important that the time and the date are correctly set in your Amiga.

To make this somewhat clearer, let's look at how make would be used to compile your editor. For this we type the names of the files which comprise your editor:

```
Editor
src/Editor.o
src/Editor.c
pre/Editor.pre
pre/Editor.prelist
```

The include files also belong in this category, but they already appear in pre/Editor.pre. The finished editor program depends on the object module src/Editor.o. The libraries also play a role, but you don't change them. You need the dependency of each file to configure the makefile in case you make any changes in the process of program development.

The object module src/Editor.o depends on the source text src/Editor.c and on the precompiled includes pre/Editor.pre; these depend on pre/Editor.prelist, which contains the names of the includes that should be precompiled. When you want to document the dependence for the make utility, enter the filenames in

the first column of a line, followed by a colon. Following that enter the filenames on which this file depends. For this first editor the dependence looks like the following:

```
Editor:          src/Editor.o
src/Editor.o:    src/Editor.c pre/Editor.pre
pre/Editor.pre: pre/Editor.prelist
```

The files `scr/Editor.c` and `pre/Editor.prelist` depend on other smaller files. So that `make` knows how to compile or link all of these files, you must enter the instructions in the `make` file. Type these directly under the line that describes the dependence. The commands cannot begin in the first column, because this is reserved for the dependents. You should also have at least one line of instructions after each one. You can enter any instructions behind each dependent. The `make` utility executes these until it either encounters a dependent or the end of the `make` file.

Linking

Now enter the instructions that link your editor created together. To make the finished program from the object modules, the following command must be executed:

```
ln src/Editor.o -lc -o Editor
```

For compilation of the source text:

```
cc +Ipre/Editor.pre src/Editor.c
```

To create the precompiled include you add two instructions. When you create these, as you did above, the C compiler creates an object file, although this wouldn't be necessary because you are only interested in the precompiled includes. The assembler is also called, which uses additional time. Combine the call of the assmebler with the `-A` option. The assembler file which the compiler created is written to the RAM disk and then deleted:

```
cc -A +Hpre/Editor.pre pre/Editor.prelist -O ram:Ed.asm
delete ram:Ed.asm
```

Your `makefile` for the creation of the editor should look like the following, yours may differ slighlty depending on the structure of your directories (i.e is the `pre` directory in the `src` directory):

```
Editor: src/Editor.o
  ln src/Editor.o -lc -o Editor

src/Editor.o: src/Editor.c pre/Editor.pre
  cc +Ipre/Editor.pre src/Editor.c

pre/Editor.pre: pre/Editor.prelist
  cc -A +Hpre/Editor.pre pre/Editor.prelist -O ram:Ed.asm
  delete ram:Ed.asm
```

Make creates the file whose name is found first in the makefile, in our case Editor. If we want to create another file (e.g., src/Editor.o) state the name when calling make from the CLI:

```
make src/Editor.o
```

Let's use make just a bit more. Because our program consists of modules, we can create the makefile before we create all of the modules. First we want the object module, from which our editor is made, to be defined in a variable:

```
OBJ=src/Editor.o
```

The variable must be directly at the beginning of the line, followed by an equal sign and the listing of the object module. This has another purpose besides forming the makefile. Because you'll want to use the debugger later to check your program for errors, you should link the program with the -w option. This option ensures that when you use the debugger, you can access the names of the functions and global variables (which simplifies the debugging). Now you have two link instructions in your makefile, from which the names of the object modules emerge. The Editor creates a link instruction while the Debug joins a symbol table with the -w option. We use the variable OBJ so we only need to define the object module once and enter only the variable to the link instructions:

```
OBJ=src/Editor.o

Editor: $(OBJ)
   ln $(OBJ) -lc -o Editor

Debug: $(OBJ)
   ln -w $(OBJ) -lc -o Editor
```

To be able to use the variable, it must be entered in parentheses with a "$" preceding the parentheses. Be sure that the same program is created by both link instructions, and that you have not forgotten a module for the instruction.

When you want to link the editor with the -w option, you call make as follows:

```
make Debug
```

The make utility determines that the file Debug doesn't exist and directs all of the steps to the creation of file specified in the makefile.

But now we come back to the 20 modules which comprise your program. If you want to state how this is created from the source text for every object module, you waste a lot of memory and create unnecessary work. To prevent this, make uses rules. One rule states

how a file with the same name and a different extension (.o) is created from a file with another extension (.c):

```
.c.o:
    /* Call compiler  c to o*/
```

This process can also be exchanged. The method is different from stating the dependents. The first extension, which must begin with a period (.), is written at the beginning of the line. The second extension follows directly behind it, and it also must begin with a period. The line ends with a colon.

The following lines list the instructions for putting the dependents to use. Because you can't directly state filenames with the rules, `make` offers two variables, both of which contain the filenames that should be used for this rule. `$@` is equivalent to the entire filename and `$*` is the name of the file without an extension or pathname. The filename has the same extension as the second extension of the rule. Take `src/Editor.o` as an example. `$@` = "`src/Editor.o`" and `$*` = "`Editor`". The following rule states a `pre` file is created from a `.prelist` file:

```
.prelist.pre:
   cc -a -o ram:$*.preasm +H$@ pre/$*.prelist
   delete ram:$*.preasm
```

The difference for the above from defined dependent is that the file that the RAM disk creates has the name `Editor.preasm` when the file `pre/Editor.pre` is be created. Now you can remove the instructions that follow your dependents for the `Editor.pre` file.

The next rule that we want to define states how an object file is created from a source text. Here you should ensure that your program contains only a main module in which the `main()` function is defined, if it consists of multiple modules. It allows the C compiler to save some memory by setting the `+B` flag for all other modules. While `main` begins with a jump to the initialization routine, the other modules do not require this jump; the initialization routine must only be at the beginning of `main`. We will define the rule so that it can create all of the modules out of the main module:

```
.c.0:
     cc +B +Ipre/Editor.pre -O $@ src/$*.c
```

Your main module represents the exception to the rule. It is sufficient to write a corresponding instruction behind the dependent definition for the `src/Editor.o` module. Then if `make` doesn't find an instruction for a dependent, it uses the corresponding rule. Your `makefile` is now finished and looks like the following listing:

```
<makefile>

.c.o:
   cc +B +Ipre/Editor.pre -O $@ src/$*.c

.prelist.pre:
   cc -A -O ram:$*.preasm +H$@ pre/$*.prelist
   delete ram:$*.preasm

OBJ=src/Editor.o

Editor: $(OBJ)
   ln $(OBJ) -lc -o Editor

Debug: $(OBJ)
   ln -w $(OBJ) -lc -o Editor

pre/Editor.pre: pre/Editor.prelist
src/Editor.o: src/Editor.c pre/Editor.pre
   cc +Ipre/Editor.pre src/Editor.c
```

Here is another file for the editor that will be used in developing our editor. The file defines the editor's data structures:

```
<src/Editor.h>

/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>

/*******************
 *
 * Data structures:
 *
 *******************/

struct Zline
{
   struct Zline *succ;
   struct Zline *pred;
   UBYTE flags;
   UBYTE len;
};

#define ZLF_USED 128
```

This file contains only the definitions for lines that you will change. The `makefile` must also be changed accordingly:

```
<makefile>
.c.o:
   cc +B +Ipre/Editor.pre -O $@ src/$*.c

.prelist.pre:
   cc -A -O ram:$*.preasm +H$@ pre/$*.prelist
   delete ram:$*.preasm

OBJ=src/Editor.o

Editor: $(OBJ)
   ln $(OBJ) -lc -o Editor

Debug: $(OBJ)
   ln -w $(OBJ) -lc -o Editor

pre/Editor.pre: pre/Editor.prelist src/Editor.h
src/Editor.o: src/Editor.c src/Editor.h pre/Editor.pre
   cc +Ipre/Editor.pre src/Editor.c
```

Editor.h is used at the moment from Editor.pre and Editor.o. To understand better what happens when make is called, enter Editor.c, Editor.h, Editor.prelist and the makefile. Now you must copy make to your disk and call make editor. The make utiliy then establishes that Editor depends on src/Editor.o, which doesn't yet exist. Make notices that it must create src/Editor.o and determines that this depends on src/Editor.c, src/Editor.h and pre/Editor.pre. While the first two files already exist, the third doesn't. Make also notices that it should create pre/Editor.pre and shows on what these depend. pre/Editor.pre depends on pre/Editor.prelist and src/Editor.h. Because both of these exist, make can create the file pre/Editor.pre following the rule .prelist.pre, in which the following command is sent to the CLI:

```
cc -A -O ram:Editor.preasm +Hpre/Editor.pre
pre/Editor.prelist
```

The unnecessary file ram:Editor.preasm can be deleted:

```
delete ram:Editor.preasm
```

After pre/Editor.pre exists, make creates src/Editor.o and executes the following command:

```
cc +Ipre/Editor.pre src/Editor.c
```

There is also a rule to make an object file from a *.c file, but because we have given an extra instruction, the rule is ignored. To conclude, the editor itself is created after the object file exists:

```
ln src/Editor.o -lc -o Editor
```

With this you have not only finished the first version of your program, but you also have a stable base on which you can build the additional programming of the editor. In large programming projects the `make` utility and the `makefile` can save a great deal of time and effort. Be sure that you understand how the `make` utility functions or see your C manual for more information on the `make` utility that came with your compiler.

4.3 Step by Step

This section describes the step by step construction of our editor program from the preceding example `Editor.c` source text. On the optional diskette each version of the editor is contained in its own directory, you may wish to do the same with your editor.

4.3.1 Opening a Window

First the editor should open a window. You could copy one of the example programs from Chapter 2 and edit it to suit your needs. Because you want to be able to open more windows, you must write your own routines to open a window and add this to the `Editor` structure. You need a function that closes a window again and releases the `Editor` structure.

Next let's consider how to open a window. This function should first try to obtain memory for the `Editor` structure. If it succeeds, a window opens and the pointer to this window is entered in the structure. Then the structure must be initialized, especially for the memory block and the line lists. The function for closing the window closes the window then releases the memory occupied by the `Editor` structure.

That would theoretically be all you need. However, if you did it this way, you would have to make some changes when you wanted to add more windows. Every window has a message port through which it sends its messages to the operating system. Because the `IntuiMessage` structure also determines which window receives the message, the message port will be enough for our purposes. Using the message port saves you memory.

Your own MessagePort

You must also make it clear to Intuition that you want a single message port for all of your windows. The program must clear the `IDCMP` flags in the `NewWindow` structure before calling `OpenWindow`. Because the window should contain no input, it receives no message port from Intuition. If a window opens, you enter the address of your message port, which you initialized from `CreatePort`, in the `UserPort` field of the `Window` structure. Then call `ModifyIDCMP` and define the input that you want to get. Because Intuition has already found a message port for the window, it doesn't use a new one. You must test for the closing of the window and set the `UserPort` field of the window structure to zero before you call `CloseWindow`, so your message port will be closed.

Key display

The next step of our editor will be to display the pressed key. The operating system has two possibilities: VANILLAKEY and RAWKEY. While you get the ASCII code of the corrseponding key with VANILLAKEY, you get scan codes with RAWKEY. RAWKEY returns which key was pressed, and not which letter the key corresponds to. Work with RAWKEY in your editor because we can determine whether the function or cursor keys were pressed. These are not passed by VANILLAKEY because these keys don't send an ASCII code.

First complete the structure that you need for your editor, this is entered in Editor.h:

```
<src/Editor.h>
/************
 * Version 1A
 * Includes:
 *
 ************/

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

/***********
 *
 * Defines:
 *
 ***********/

#define MAXWIDTH 80

/*******************
 *
 * Data structures:
 *
 *******************/

struct Zline
{
  struct Zline *succ;
  struct Zline *pred;
  UBYTE flags;
  UBYTE len;
};

#define ZLF_USED 128

struct ZList
{
  struct Zline *head;
  struct Zline *tail;
  struct Zline *tailpred;
};

struct Memorysection
{
  struct Memorysection *succ;
  struct Memorysection *pred;
  UWORD len;
};
```

```
struct FList
{
  struct Memorysection *head;
  struct Memorysection *tail;
  struct Memorysection *tailpred;
};

struct Memoryblock
{
  struct Memoryblock *succ;
  struct Memoryblock *pred;
  ULONG length;
  ULONG free;
  struct FList freeliste;
};

struct BList
{
  struct Memoryblock *head;
  struct Memoryblock *tail;
  struct Memoryblock *tailpred;
};

struct Editor
{
  struct Editor *succ;
  struct Editor *pred;
  struct Window *window;
  struct BList block;
  struct ZList zlines;
  UBYTE buffer[MAXWIDTH];
  UBYTE tabstring[MAXWIDTH];
  UWORD num_lines;
  struct Zline *actual;
  struct Zline *top;
  UWORD toppos;
  UWORD xpos,ypos;
  UWORD changed:1;
  UWORD insert:1;
};

struct EList
{
  struct Editor *head;
  struct Editor *tail;
  struct Editor *tailpred;
};
```

The structures in the above code have already been described in Section 4.1, except for the last one. The `EList` structure manages all of the editor windows, just as the `BList` structure manages all of the memory blocks and the `ZList` structure manages all of the lines. Next follows the source text of the editor:

```
<src/Editor.c>

/* Editor.c V1A */
/************
 *
 * Includes:
 *
```

```
 ************/
#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

#define REV 33L

/**********************
 *
 * External Functions:
 *
 **********************/

struct Library *OpenLibrary();
struct Window *OpenWindow();
void CloseLibrary(),NewList(),AddTail(),CloseWindow(),free();
void DeletePort(),ModifyIDCMP(),ReplyMsg();
struct Editor *malloc(),*RemHead();
struct MsgPort *CreatePort();
struct IntuiMessage *GetMsg();
ULONG Wait();

/**********************
 *
 * Global Variables:
 *
 **********************/

struct Library *IntuitionBase = NULL, *GfxBase = NULL, *DosBase =
NULL;


struct EList editorList;

struct NewWindow newEdWindow =
{
  100,40,440,156,
  AUTOFRONTPEN,AUTOBACKPEN,
  REFRESHWINDOW | MOUSEBUTTONS | RAWKEY | CLOSEWINDOW,
  WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH |    WINDOWCLOSE |
SIZEBBOTTOM
  | SIMPLE_REFRESH | ACTIVATE,
  NULL,NULL,
  (UBYTE *)"Editor",
  NULL,NULL,
  100,40,640,200,
  WBENCHSCREEN
};

struct MsgPort *edUserPort = NULL;


/***************
 *          *
 * Functions: *
 *          *
```

```
***************/

/**********************************
*
* OpenEditor()
*
* Open Editor window and
* initalize Editor structure
*
* Returns pointer to Editor structure
* or is zero in case of an error
*
**********************************/

struct Editor *OpenEditor()
{
  register struct Editor *ed = NULL;
  register struct Window *wd;
  register ULONG flags;

  /* IDCMPFlags saved, sets it to zero in structure
    => use your own UserPort! */
  flags = newEdWindow.IDCMPFlags;
  newEdWindow.IDCMPFlags = NULL;

  /* Memory for editor structure: */
  if (ed = malloc(sizeof(struct Editor)))

    /* Window open: */
    if (wd = OpenWindow(&newEdWindow))
    {
      ed->window = wd;

      /* UserPort established: */
      wd->UserPort = edUserPort;
      ModifyIDCMP(wd,flags);

      /* Parameter initialization: */
      NewList(&(ed->block));
      NewList(&(ed->zlines));
      ed->num_lines = 0;
      ed->actual = NULL;
      ed->top = NULL;
      ed->toppos = 0;
      ed->xpos = 1;
      ed->ypos = 1;
      ed->changed = 0;
      ed->insert = 1;
    }
    else
    {
      free(ed);
      ed = NULL;
    }

  newEdWindow.IDCMPFlags = flags;
  return (ed);
}

/***************************
*
* CloseEditor(ed)
```

```
 *
 * Close editor window.
 *
 * ed ^ Editor structure.
 *
 ***************************/

void CloseEditor(ed)
struct Editor  *ed;
{
  /* Close window: */
  ed->window->UserPort = NULL;
  CloseWindow(ed->window);
  free(ed);
}

/*****************
 *
 * Main program:
 *
 *****************/

main()
{
  register struct Editor *ed;
  BOOL running = TRUE;
  register struct IntuiMessage *imsg;
  ULONG signal,class;
  UWORD code,qualifier;
  APTR iaddress;
  register UWORD n = 1;

  /* Libraries open: */

  if ( !(IntuitionBase = OpenLibrary("intuition.library",REV)))
    goto Ende;
  if ( !(GfxBase = OpenLibrary("graphics.library",REV)))
    goto Ende;
  if ( !(DosBase = OpenLibrary("dos.library",REV)))
    goto Ende;

  /* UserPort open */
  if (! (edUserPort = CreatePort(NULL,0L))) goto Ende;

  /* Editor list initialization: */
  NewList(&editorList);

  /* Open first editor window: */
  if (ed = OpenEditor())
     AddTail(&editorList,ed);
  else
    goto Ende;

  do
  {
    signal = Wait(1L << edUserPort->mp_SigBit);

    while (imsg = GetMsg(edUserPort))
    {
      class    = imsg->Class;
      code     = imsg->Code;
      qualifier = imsg->Qualifier;
```

```
        iaddress  = imsg->IAddress;

        ReplyMsg(imsg);

        /* Event processing: */
        switch (class)
        {
          case RAWKEY:
          printf("RawKey-Event Number %3d code: Code = $%4x,
Qualifer = $%4x.\n",n++, code, qualifier);
            break;

          case CLOSEWINDOW:
            running = FALSE;
            break;

          default:
            printf("Not a processable event: %lx\n",class);

        } /* of case */
      } /* of while (GetMsg()) */
    } while (running);

Ende:
  /* Close all editor windows: */
  while (ed = RemHead(&editorList))
    CloseEditor(ed);

  /* Close UserPort: */
  if (edUserPort) DeletePort(edUserPort);

  /* Close Libraries: */
  if (DosBase) CloseLibrary(DosBase);
  if (GfxBase) CloseLibrary(GfxBase);
  if (IntuitionBase) CloseLibrary(IntuitionBase);
}
```

The `OpenEditor` function works exactly as described. First open the libraries in the main program, then a single `MessagePort` (`edUserPort`) is entered in the user port of the window with `OpenEditor`. Then the editor list is initialized and a window opens and is connected to the editor list (`AddTail`). The main loop follows, which waits for input in the main loop (`do {...} while (running)`). The operating system signals an event. Messages are received until no more are present; it could be that the editor only received one signal, but multiple messages. In the `switch case` function they are differentiated according to the message type and handled accordingly. In this case with `RAWKEY` only the code of the pressed key is displayed, a number (n) counted, `CLOSEWINDOW` set to FALSE and the program ended. To conclude all of the windows must be closed again, then the message port and lastly the libraries.

A few observations about the program:

- To write the program to run on all compilers, you should use the variable types from <exec/types.h> and not the normal C

standard data types, because the data type integer is implemented once as a 16-bit number and once as a 32-bit.

- When you want to save casting (type conversion), you can declare the functions as you want. Especially functions, that the pointer to objects returns, can be declared so that the pointer returns all of the possibilties, like it did in the program with `malloc`, `RemHead` and `GetMsg`.

- You should also declare all of the functions that you use in your program, if only as `void`. Otherwise you could easily overlook a function, and the program could stop because you needed a 32-bit pointer but only had a 16-bit pointer available. There is also no problem if you must compute or return with a 32-bit integer instead of with a 16-bit integer.

- The `goto` statement in conjunction with the error handler is better than `IF` statements. The function listed below, which could follow every error in the main program, is very inefficient compared to just using `goto`:

```
if (error)
{
   CloseWindow(xxx);
   CloseLibrary(yyy);
   exit(FALSE);
}
```

- The functions `NewList`, `AddTail` and `RemHead` are for `Exec` lists. When you look at the structures of lists and nodes (see the Appendix for a description of `Exec`), you determine that these are only minimally more extensive than your structures. The functions named work with your structures because they don't access additional elements. It would be different if you also used the `Enqeue` function because this also uses the priority field of the `Exec` node, which your structure does not have.

Compile and link the program. Run it and press a couple of keys. The program returns a message when you press a key, and one when you release a key. When you press one key and hold it down, something unexpected happens: Once a certain amount of time passes (set by `KeyRepeatDelay`), the program displays the same message multiple times. This continues until you release the key. The only difference between these messages is that the ninth bit of the qualifier is set, which means that this key transmits the key repeat function. The editor allows key repeat with both `VANILLAKEY` and `RAWKEY` modes.

4.3.2 From RAWKEY to ASCII

The question remains how we can convert RAWKEY characters into normal ASCII characters. A user-defined conversion table is out of the question, because this won't work on Amigas using alternate keyboard maps. The operating system also has a function that handles keyboard layout.

The RawKeyConvert function is part of the console device. This function can be accessed by selecting a pointer to console base without opening the console device first, because the function can be called like a library function. You receive the pointer to the console base by calling:

```
OpenConsole("console.device",-1L,IOStd,0L);
```

and selecting the pointer from the io_Device field of the IOStdReq structure. Then you can call RawKeyConvert as a normal function. In addition to a pointer to an InputEvent structure, you must assign a pointer to a buffer whose length and a pointer to a KeyMap structure has been set to zero, because the standard keyboard table should be used for the conversion. You must fill the Inputevent structure with the RAWKEY code and the qualifier from the IntuiMessage structure before the call. To allow ASCII output instead of RAWKEY output, the followings changes are necessary:

Define the maximum length of keyboard input as 128 characters:

```
#define MAXINPUTLEN 128L
```

Declare the new OpenDevice and RawKeyConvert functions:

```
ULONG OpenDevice();
SHORT RawKeyConvert();
```

Now two global variables appear that we already mentioned. We set the ie_Class field to RAWKEY in the InputEvent structure so that you don't have to do this explicitly in the program:

```
struct IOStdReq ioStdReq;

UBYTE inputBuffer[MAXINPUTLEN];
UWORD inputLen = 0;

struct InputEvent inputEvent =
{
   0,
   IECLASS_RAWKEY,0,
   0,0
};
```

The characters from the RawKeyConvert function pass to the input buffer, stored with the corresponding key. Define one variable at the beginning of the main program:

```
SHORT l;
```

After you open the libraries in the main program, select the pointer to the console device:

```
if ( !(OpenDevice("console.device",-1L,&oiStdReq,0L)))
  ConsoleDevice = ioStdReq.oi_Device;
else
  goto Ende;
```

Finally, construct the handling of RAWKEY messages in the main loop:

```
case RAWKEY:
   if (! (code & IECODE_UP_PREFIX))
   {
      inputEvent.ie_Code  = code;
      inputEvent.ie_Qualifier = qualifier;
      if ((l = RawKeyConvert(&inputEvent,
                  inputBuffer,MAXINPUTLEN,NULL)
            ) > 0)
      {
       inputBuffer[l] = 0;
       printf("%3d> Length = %d: ",n++,l);
       for (l = 0; inputBuffer[l]; l++)
          printf("%2x",(UWORD)inputBuffer[l]);
       if (inputBuffer[0] >= 32 && inputBuffer[0] <= 127)
            printf(" %s",inputBuffer);
         printf("\n");
      }
   }
   break;
```

First test whether the corresponding key is pressed or released; the second case (IECODE_UP_PREFIX) isn't currently of interest. Then the values from code and qualifier transfer into the InputEvent structure. RawKeyConvert is then called. Save the return value in l (not in inputLen) and test whether this value is greater than 0. If l is less than 0, an error occurs (buffer too small). If l equals zero, the key returns no characters.

If the returned value is greater than zero, it gives the length of the string stored with the key. A null byte appears behind the string so that you can output the string with printf. Normally keys are assigned a single character so the value one is usually returned as the length, but there are exceptions. The program returns a successive number and then the length of the string. Then follows the string itself, first in hex notation then as a true ASCII string, but only if the first character lies between a space (32) and a delete (127), and is a printable character. If you don't encounter this check, your output appears in the CLI window because the CLI converts certain ASCII codes to cursor movements.

Type it in, compile, link, run and examine. While testing the normal keys don't give any unusual results, the function keys return three or four characters at a time. You must bear this in mind with your keyboard check. This is also the reason that VANILLAKEY returns no function key information. The IntuiMessage can only return one character, not three at once. When you examine all of the special characters and compare them with the codes supplied in the documentation (see the AmigaDOS manual), you may find that some codes may not match. There may be some errors in your output.

The complete source text for the section of the editor described above can be found in the V0.1 directory on the optional disk for this book. The complete listing for readers who do not have the optional disk is presented in Appendix A. If you own an Aztec C compiler you can compile this by changing to the V0.1 directory and entering make Editor. Before this you must ensure that the make utility exists in a place accessibe to the CLI. For example, you could copy make to the RAM disk:

```
copy make to ram:
path ram: add
```

Enter the following to compile the program on the optional diskette:

```
cd V0.1
make Editor
```

The single difference from the paths discussed here is that the pre directory is elsewhere on the disk, in the main editor directory. Because pre is used for all of the versions of the editor, you should make sure that the file :pre/Editor.pre exists for each new version of the editor. Delete the file pre/Editor.pre before you call make for each new version.

4.3.3 Memory Management

You must have a method of saving and joining lines together before you can load and display a file. You need a function for this which provides a new Zline structure, into which you can copy a string and the current line. The function header should look like the following:

```
struct ZLine *newZline(len)
UWORD len;
```

len represents the length of the string, including the end of line character. Because you can only use memory pieces of even lengths, the length must appear in the function as an even number. Because this happens often, create a corresponding define:

```
#define EVENLEN(x) (((X)+1)&-2)
```

Before you implement the function, consider how this should operate. If no memory block is found in the memory list and the text memory is also empty, a new block with 10K appears and a line is selected. Otherwise a search begins for an available line. If this happens, a garbage collection executes and the memory is released before a second search executes. If this also happens, not enough memory blocks exist for further line storage. The program uses a new memory block of only 5K. That is why you should work with two different block sizes, so that only one memory block will be needed for texts less than 10K in length. The following flowplan explains our method:

```
If no memory block exists in the block list
  select a new block (length = 10K)
  select line
  done
else
  select line
  if none is found
    execute garbage collection
    select line
    if none is found
      select new block (5K)
      select line
  done
```

The next function uses a new memory block:

```
struct MemoryBlock *newBlock(len)
ULONG len;
```

If a new memory block can be used, this memory block structure is initialized and a pointer returns to the block, otherwise a null is returned for the block. The memory block must then be inserted in the memory block list. For this you need a pointer to the current `Editor` structure:

```
struct Editor *actualEditor;
```

The memory block accesses `AllocMem` instead of the standard C function `malloc`. This route is actually better than using `malloc`, because `malloc` places the used memory in a list so that it can release all memory not specifically released at the end of the program. This list uses memory. Since all of the memory should be released before the end of the program anyway, use `AllocMem`.

The same reasons apply for not using the `AllocRemember` function, which the Intuition library provides for memory management. The advantage to this function is its recall of all memory that you allocated allowing easy release. The `Remember` structure must be saved with `AllocRemember`, which uses additional memory.

The functions presented so far have needed little explanation. This next function requires some description, however:

```
struct Zline *getZline(len)
UWORD len;
```

This function searches all the memory blocks of the current editor for a memory piece that can accept the line. The line length is given in `Line`, which must be an even value within the function. It searches for a memory block, then determines which block is large enough to accept the text. Since you have not added memory blocks, we can distribute the memory pieces in a more optimal way:

First try to find a memory piece that has the exact length of the text. If this doesn't happen, search for the largest memory piece available. This avoids making a small piece smaller. If a memory piece is so small that after the new line is cut from it, not enough memory exists for the memory piece structure, and at least two bytes remain, the search was unsuccessful. Because the rest must be at least as large, you can insert another line there.

Remember that each memory block consists of two things: The lines that lie in it, and the free memory pieces from which room for new lines is taken.

The following function is the last accessed from `newZline`:

```
struct Memoryblock *garbageCollection(len)
UWORD len;
```

This function reorganizes the free memory inside of the memory blocks so that only one memory piece exists with the maximum size per memory block. Each memory piece needs ten bytes of management information (`sizeof(Memorysection)`). Multiple memory pieces in a memory block allocate a little free memory. Garbage collection executes memory block by memory block, until a memory block with a free memory piece the size of `len` bytes is found, or until the end of the memory block list occurs. If the search was successful, a pointer to the corresponding memory block returns (otherwise, zero is returned). The pointer can eventually be used to keep the `GetZline` function from searching through every memory block.

All of these functions appear in a module called `Memory.c`. The entire module is in the following listing:

```
<src/Memory.c>

/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include <exec/memory.h>
#include "src/Editor.h"
```

```
/***********
 *
 * Defines:
 *
 ***********/

#define BIGBLOCK (200L-sizeof(struct Memoryblock))
#define BLOCKSIZE (100L-sizeof(struct Memoryblock))

/**********************
 *
 * External Functions:
 *
 **********************/

struct Memoryblock *AllocMem();
void NewList(),AddTail(),Remove(),FreeMem();

/*********************
 *
 * External Variables:
 *
 *********************/

extern struct Editor *actualEditor;

/***************
 *             *
 * Functions:  *
 *             *
 ***************/

/*****************************************
 *
 * freeMemoryblock(blk)
 *
 * frees the memory of memory
 * blocks free, without regard for
 * lines that lie in it.
 *
 * blk ^ Memoryblock.
 *
 *****************************************/

void freeMemoryblock(blk)
struct Memoryblock  *blk;
{
  Remove(blk);
  FreeMem(blk,blk->length + sizeof(struct Memoryblock));
}

/*********************************************
 *
 * newerBlock(len):
 *
 * used memory for new memory block with len length
 * and returns the pointer
 * in case of error => NULL.
 *
 * len = Length of memory block.
 *
```

```
 *********************************************/

struct Memoryblock *newerBlock(len)
ULONG                    len;
{
  register struct Memoryblock *blk;
  register struct Memorysection *spst;

  /* Allocate memory: */
  if (blk = AllocMem(len+sizeof(struct Memoryblock),MEMF_CLEAR))
  {
    /*Initialize memory block structure: */
    blk->length = len;
    blk->free = len - sizeof(struct Memorysection);
    NewList(&(blk->freeliste));
    spst = (struct Memorysection *)(blk + 1);
    AddTail(&(blk->freeliste),spst);
    spst->len = (UWORD)blk->free;
  }

  return (blk);
}

/*********************************************
 *
 * searchMemorysection(block,len)
 *
 * Searches for memory piece with len length in the block block
 * and returns the pointer or zero in case
 * of no room.
 *
 * block ^ Memoryblock.
 * len = Length of line (even).
 *
 *********************************************/

struct Memorysection *searchMemorysection(block,len)
struct Memoryblock                *block;
UWORD                              len;
{
  register UWORD minl,l;
  register struct Memorysection *st,*fnd = NULL;

  /* Now searches through if enough memory: */
  if ((ULONG)len <= block->free)
  {
    /* Minimal length in case found Length <> len */
    minl = len + sizeof(struct Memorysection) + 2;

    /* Searches through list: */
    for (st = block->freeliste.head; st->succ; st = st->succ)
      if ((l = st->len) == len)
      {
        /* Optimal size found: */
        fnd = st;
        break;
      }
      else
        if ((l >= minl) && (l > ((fnd)? fnd->len : 0)))
          /* Search for largest block: */
          fnd = st;
  }
```

```
  return (fnd);
}
/*****************************************
 *
 * ConvertSpstToZline(blk,st,len)
 *
 * Converts memory piece in line.
 * If the piece was longer than the line,
 * the piece is shortened, otherwise the piece is removed
 * from the list.
 *
 * blk ^ memory block of memory piece.
 * st ^ memory piece.
 * len = Length of the line
 *
 *****************************************/

struct Zline *ConvertSpstToZline(blk,st,len)
struct Memoryblock          *blk;
register struct Memorysection   *st;
register UWORD                   len;
{
  register UWORD l;
  register struct Zline *z;

  if (st->len == (l = EVENLEN(len)))
  {
    /* Found piece fits exactly: */
    Remove(st);
    z = struct Zline *)st;
  }
  else
  {
    /* .distribute piece: */
    z = (struct Zline *)
    ((UBYTE *)st) + sizeof (struct  Memorysection) + st->len - (l
+= sizeof (struct Zline)) );
    st->len -= l;
  }
  blk->free -= l;

  z->flags  = 0;
  z->len  = len;

  return (z);
}

/*********************************************
 *
 * getZline(len)
 *
 * Tries to get line with len length and,
 * searches through all of the memory blocks.
 * returns pointer to line or
 * zero if no room.
 *
 * len = Length of the line.
 *********************************************/
struct Zline *getZline(len)
UWORD             len;
{
```

```
  register UWORD l;
  register struct Memoryblock *blk,*fblk = NULL;
  register struct Memorysection *st,*fst = NULL;

  /* make even length: */
  l = EVENLEN(len);

  /* run through list of blocks: */
  for (blk = actualEditor->block.head; blk->succ; blk = blk-
>succ)
    if (st = searchMemorysection(blk,l))
      if (st->len == l)
      {
        /* found optimal memory piece: */
        fblk  = blk;
        fst = st;
        break;
      }
      else
        if (st->len > ((fst)? fst->len : 0))
        {
          /* searches for largest memory piece: */
          fblk  = blk;
          fst = st;
        }

  if (fst)
    return (ConvertSpstToZline(fblk,fst,len));
  else
    /* no piece found: */
    return (NULL);
}

/*******************************************
 *
 * optimalBlock(block)
 *
 * Searches through all of the memory pieces of the block
 * and makes one out of all of the memory pieces
 * that lie one after another.
 *
 * block ^ Memoryblock.
 *
 *******************************************/

void optimalBlock        (blk)
register struct Memoryblock *blk;
{
  register struct Memorysection *st1,*st2,*st1e;

  for (st1 = blk->freeliste.head; st1->succ; st1 = st1->succ)
  {
    /* Certain pointer directly behind st1: */
    st1e = (struct Memorysection *)
         (((UBYTE *)st1) + st1->len + sizeof(struct
Memorysection));

    for (st2 = blk->freeliste.head; st2->succ; st2 = st2->succ)
      if (st1e == st2)
      {
        /* memory piece st2 lies behind st1: */
        st1->len += st2->len + sizeof(struct Memorysection);
```

```
        blk->free += sizeof(struct Memorysection);
        Remove(st2);
        st1e = (struct Memorysection *)
             (((UBYTE *)st1)+st1->len+sizeof(struct
Memorysection));

        /* begin at the beginning of the list: */
        st2 = (struct Memorysection *) &(blk->freeliste.head);
      }
  }
}

/*************************************
 *
 * garbageCollection(len)
 *
 * cleans the memory until at least
 * len bytes are free. returns pointer.
 * to memory block where there is room
 * or zero if no room.
 * zero if no room.
 *
 * len = number of bytes that should be accomodated
 *
 *
 *************************************/

struct Memoryblock *garbageCollection(len)
UWORD  len;
{
  struct Memoryblock *blk;
  struct Memorysection *st,spst;
  register UBYTE *ptr,*end;
  register struct Zline *z,*fz;
  register UWORD n;

  for (blk = actualEditor->block.head; blk->succ; blk = blk-
>succ)
    /* Only garbage collection if more than one memory piece! */
    if ((st = blk->freeliste.head->succ)? st->succ : FALSE)
    {
      /* test for beginning and end of block memory: */
      ptr = (UBYTE *)(blk + 1);
      end = ptr + blk->length;

      /* lines lie at the beginning: */
      while (ptr < end)
      {
        /* searches for lines that lie at next ptr: */
        fz = NULL;
        for (z = actualEditor->zlines.head; z->succ; z = z-
>succ)
          if ((z >= ptr) && (z < end))
            if ((z < fz) || (fz == NULL))
              fz = z;

        if (fz)
          if (fz != ptr)
          {
            /* note old pointer: */
            z = fz;
            st = (struct Memorysection *)ptr;
```

```
            spst = *st;

            /* writes line after ptr: */
            for (n = (EVENLEN(fz->len) + sizeof(struct Zline))
>> 1;
                n; n--)
              *((UWORD *)ptr)++ = *((UWORD *)fz)++;

            /* pointer correction: */
            fz = (struct Zline *)st;
            fz->succ->pred = fz;
            fz->pred->succ = fz;
            if (actualEditor->actual == z)
              actualEditor->actual = fz;
            if (actualEditor->top == z)
              actualEditor->top = fz;

            st = (struct Memorysection *)ptr;
            st->succ = spst.succ;
            st->pred = spst.pred;
            st->len = spst.len;
            spst.succ->pred = st;
            spst.pred->succ = st;

            /* push memory pieces together: */
            optimalBlock(blk);
          }
          else
            /* set ptr high: */
            ptr += EVENLEN(fz->len) + sizeof(struct Zline);
        else
          /* no more lines in block! */
          break;
      } /* of while */
      /* Test if there is enough memory: */
      if (searchMemorysection(blk,len))
        return (blk);
    } /* of if (at least 2 blocks) */

  return (NULL);
}

/*****************************************
 *
 * newZline(len)
 *
 * get room for new line,
 * fits characters in len.
 * returns pointer to new line,
 * or zero if error.
 *
 * len = number of characters.
 *
 *****************************************/

struct Zline *newZline(len)
UWORD             len;
{
  register struct Zline *z;
  register struct Memoryblock *blk;
  register struct Memorysection *st;
  register UWORD l;
```

```
  if (actualEditor->block.head->succ)
    /* blocks already exist: */
    if (z = getZline(len))
      return (z);
    else
      /*  Garbage-Collection:line not directly usable */
      if (blk = garbageCollection(l = EVENLEN(len)))
        if (st = searchMemorysection(blk,l))
          return (ConvertSpstToZline(blk,st,len));
        else
          /* may not actually be encountered !!! */
          return (NULL);
      else
        /* no reults for Garbage-Collection ->pick new block */
        if (blk = newerBlock(BLOCKSIZE))
        {
          AddTail(&(actualEditor->block),blk);
          return (getZline(len));
        }
        else
          return (NULL);
  else
    /* no block in the block list: */
    if (blk = newerBlock(BIGBLOCK))
    {
      AddTail(&(actualEditor->block),blk);
      return (getZline(len));
    }
    else
      return (NULL);
}
```

Some functions appear in the above code that haven't yet been described. This is because some jobs can be done better with functions, and you don't have to know all the details on every function. Here are a couple of words for the additional functions:

```
void freeMemoryblock(blk)
struct Memoryblock *blk;
```

This function releases the memory occupied by `Memoryblock blk`. There is no consideration if other lines lie within a block. `CloseEditor` uses this function to free the entire memory of the editor.

```
struct Memorysection *searchMemorysection(block,len)
struct Memoryblock                    *block;
UWORD                                        len;
```

This function searches the given block for a memory piece that either has the length `len`, or for the largest existing memory piece. Either a pointer to the found memory piece or null returns, which means none of sufficient length could be found. Be very careful that `blk->free` contains the sum of the lengths of the individual memory pieces of this block, so that a simple check determines if enough space is available. If the length could not be found, only the larger memory pieces are of interest: those (`sifeof(struct Memorysection)` +2) bytes larger

than `len`, because the found memory piece must be distributed between a line and a memory piece. When no more room exists for the structures, no distribution occurs.

The following line may be difficult to understand:

```
if ((l >= minl) && (l > ((fnd)? fnd->len : 0)))
```

For those who have trouble with the above line, here is a small tip. If `fnd` equals zero, no memory piece was found and cannot be tested to see if `l` is greater than the length of the memory pieces already found (`fnd`). In any case, the second comparison should be true. The expression `((fnd)? fnd->len : 0)` equals zero if `fnd` is zero. Otherwise `fnd->len` (the length of the memory piece `fnd`). Because `l` is always positive, `l` is always greater than `0`, letting you bypass the special case when `fnd` is still zero.

```
struct Zline *ConvertSpstToZline(blk,st,len)
struct Memoryblock                  *blk;
register struct Memorysection       *st;
register UWORD                      len;
```

When you want to use a new line, you must convert a memory piece into a line, or at least take a line from a memory piece. The function mentioned earlier lowers the amount of work required for this conversion. If the memory piece was as long as the line, it is removed from the list of all of the memory pieces. Otherwise the line is removed from the end of the memory piece with the corresponding length. In each case the number of free bytes of the block belonging to the memory piece decreases. It then returns a pointer to the line.

The following function appends memory pieces that lie directly behind one another:

```
void optimalBlock (blk)
struct Memoryblock *blk;
```

You'll always need this function when you lengthen a memory piece or insert a new memory piece in the list. You must be sure that no two memory pieces are directly behind one another, because this costs you 10 bytes (`sizeof(struct memorypiece)`).

The following flowplan describes the `garbageCollection` function:

```
FOR all blocks
  are there at least 2 memory pieces in this block?
  move all lines in this block to their beginning
  join all memory pieces into one
  is there enough room in this block?
  return the pointer to the block

else: return zero
```

4.3.4 Testing the new functions

Now we should test the functions created in the `memory` module. We will create a module used to test our newly created functions. The following `test.c` module allows you to call all of the important functions and display the occupied memory:

```
<src/Test.c>

/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

/**********************
 *
 * External Functions:
 *
 **********************/

struct Zline *newZline();
UBYTE *gets(),getchar();
UWORD strlen();
void strncpy(), AddTail(),deleteZline();

/*********************
 *
 * External variables:
 *
 *********************/

extern struct Editor *actualEditor;

/***************
 *
 * Functions: *
 *
 ***************/

void print()
{
  register struct Zline *z;

  for (z = actualEditor->zlines.head; z->succ; z = z->succ)
    printf("%s\n",z+1);
}
```

```
void print_blocks()
{
  register struct Memoryblock *blk;

  printf("Block list:\n\n");
  for (blk = actualEditor->block.head; blk->succ; blk = blk-
>succ)
    printf("Block %8lx, Lenght = %ld, Free = %ld\n",
          blk,blk->length,blk->free);

  printf("\n");
}

void input()
{
  UBYTE s[80];
  register struct Zline *z;
  register UWORD l;

  printf("New Line: ");
  if (gets(s))
  {
    l = strlen(s);
    if (s[l-1] != ' ')
    {
      s[l++] = 13;
      s[l] = 0;
    }
    if (z = newZline(l))
    {
      strncpy(z+1,s,l);
      AddTail(&(actualEditor->zlines),z);
    }
    else
      printf("Error in new line!!!\n");
  }
  else
    printf("Error with gets!!!\n");
}


void Test()
{
  register UBYTE c;

  printf(">");
  while ((c = getchar()) != 27)
  {
    while (getchar() != 10) ;    /* read one CR! */
    switch (c)
    {
      case 'a':
        input();
        break;
      case 'p':
        print();
        break;
      case 'b':
        print_blocks();
        break;
    }
```

```
    printf(">");
  }
}
```

The `print` function displays the entire text that you have entered. The `print_blocks` function specifies the starting address, line lengths and the number of free bytes of all of the memory blocks currently in the editor. `input()` expects line input that is added to the end of the list of all lines. Otherwise, as you do later in the real editor, a null byte represents the end of line, with which the line forms the output of all of the lines as simply as possible. There the `printf` function expects a pointer to a string as a parameter, which ends with a null byte.

These functions are called from the `Test` function which waits for line input from a user. You can add a line by pressing <a> (Append), print the text by pressing <p> (Print), and read a list of all of the memory blocks by pressing <b> (Blocks). After you enter a letter you must press the <Return> key, because we still use the CLI window for the input routine and it is line oriented (input will be sent only after pressing <Return>). The <Return> is not used in the `switch case`.

Now you must change your program so that the `Test` function is automatically called. Change the `Editor` module as follows. First define the external functions:

```
void Test();
void freeMemoryblock();
```

Then change the `CloseEditor` function so that all memory is released again:

```
void CloseEditor(ed)
struct Editor  *ed;
{
  register struct Memoryblock *blk;

  /* Free first memory block: */
  while (blk = (struct Memoryblock *)RemHead(&(ed->block)))
    freeMemoryblock(blk);

  /* Then Close window: */
  ed->window->UserPort = NULL;
  CloseWindow(ed->window);
  free(ed);
}
```

And in the main program add the `Test` function before the main loop and before the `Wait` function:

```
Test();
```

Finally, change the `makefile` so that the two new modules are also compiled. To do this, expand the line in which `OBJ` is defined:

```
OBJ=src/Editor.o src/Memory.o src/Test.o
```

At the end of the `makefile` insert the following two lines:

```
src/Memory.o: src/Memory.c src/Editor.h pre/Editor.pre
src/Test.o: src/Test.c src/Editor.h pre/Editor.pre
```

The last change that you should make is set the `BIGBLOCK` and `BLOCKSIZE defines` to smaller values so that you don't have to enter 10K of text later before you get a second memory block. Change 10K into 120 bytes and 5K into 60 bytes. This also lets you check what happens when you want to insert a line that is longer than an entire empty block.

Compile, link and test. When inserting a line that is larger than the block, a new block is used, even though the line cannot fit there. Because you set the block size to 5K for later, the following happens: if only 22 bytes are free in a block, a line with 21 or 22 characters fits there; or if you insert one with less than 10 characters, then there must be room for a `Memorysection` structure.

4.3.5 Deleting Lines

Until now you have only tested the functions for inserting a new line. To be able to test the functions `optimalBlock` and `garbage Collection`, you need a function that can delete any line. This function first removes the line from the list, searches for the block where it lies and converts the line into a memory piece. Then the block is optimized. The function for the editor looks like this:

```
void deleteZline      (line)
register struct Zline *line;
{
  register struct Memoryblock *blk,*fblk = NULL;
  register struct Memorysection *st;

  /* remove line from list: */
  Remove(line);

  /* set pointer to this line to zero: */
  if (actualEditor->actual == line)
    actualEditor->actual = NULL;
  if (actualEditor->top == line)
    actualEditor->top = NULL;

  /* determine in which block the line lies: */
  for (blk = actualEditor->block.head; blk->succ; blk = blk-
>succ)
    if ((line > blk) && (line < (((UBYTE *)(blk + 1)) + blk-
>length)))
    {
      fblk = blk;
      break;
    }
```

```
  if (fblk)
  {
    /* convert line into memory peices, insert in list: */
    fblk->free += ( ((struct Memorysection *)line)->len
                  = EVENLEN(line->len) );
    AddTail(&(fblk->freeliste),line);

    optimalBlock(fblk);
    if (fblk->length == (fblk->free + sizeof(struct
Memorysection)))
      /* Block contains no more lines: Delete! */
      freeMemoryblock(fblk);
  }
}
```

If the line was the last line in the memory block, this is deleted. Before you write a few additional functions for testing, a note about `ConvertSpstToLine`. Because it was easy to program, we created the function so that the line ends if the line to be inserted is smaller than the memory piece. This means that the lines in a block extend from back to front.

You have already determined that the `garbage collection` moves all the lines to the beginning of the block. It's not too late to fix this back-to-front problem. Change the `ConvertSpstToLine` function so that the lines extend from front to back:

```
struct Zline *ConvertSpstToZline(blk,st,len)
struct Memoryblock          *blk;
register struct Memorysection   *st;
register UWORD                   len;
{
  register UWORD l;
  register struct Zline *z;

  z = (struct Zline *)st;

  if (st->len == (l = EVENLEN(len)))
    /* Found piece fits exactly: */
    Remove(st);
  else
  {
    /* distribute piece: */
    ((UBYTE *)st) += (l += sizeof (struct Zline));
    ( st->succ = ((struct Memorysection *)z)->succ )->pred = st;
    ( st->pred = ((struct Memorysection *)z)->pred )->succ = st;
    st->len = ((struct Memorysection *)z)->len - l;
  }
  blk->free -= l;

  z->flags  = 0;
  z->len    = len;

  return (z);
}
```

A small disadvantage to this solution: The pointer for the chaining must be adjusted. This gives you a chance to play with the many

possibilities of C. The assembler code that changes the pointer is fairly efficient. Look at the assembler code. Compile the `Memory` module as follows:

```
cc +B +Ipre/Editor.pre src/Memory.c -a -t -o ram:sp.asm
```

The created assembler code can be found on the RAM disk under the name `sp.asm`. Because the assembler code contains the C commands as comments, it's not too difficult to find the place. Search for the label `_ConvertSpstToZline`, then cursor down until you see the above lines as comments. Directly under that you should find the assembler code for these lines. This code looks like this:

```
;  ( st->succ = ((struct Memorysection *)z)->succ )->pred = st;
;
 move.l (a3),a0       ;a3 = z
 move.l a0,(a2)       ;a2 = st;
 move.l a2,4(a0)
;  ( st->pred = ((struct Memorysection *)z)->pred )->succ = st;
 move.l 4(a3),a0
 move.l a0,4(a2)
 move.l a2,(a0)
```

Notice how the assembler code looks is similar for both functions. You can improve it. For example, the following section from the `garbageCollection` function moves a piece of memory:

```
;       /* push line down after ptr: */
;       for (n = (EVENLEN(fz->len) + sizeof(struct Zline)) >> 1;
;          n; n--)
 move.l d5,a0         ;d5 = fz
 move.l #0,d0         ;d6 = n
 move.b 9(a0),d0      ;a2 = ptr
 add.w #1,d0
 bclr.l #0,d0
 add.w #10,d0
 move.w d0,d6
 lsr.w #1,d6
 bra .74
.73
;        *((UWORD *)ptr)++ = *((UWORD *)fz)++;
 move.l d5,a0
 add.l #2,d5
 move.l a2,a1
 add.l #2,a2
 move.w (a0),(a1)
.71
 sub.w #1,d6
.74
 tst.w d6
 bne .73
.72
;
```

Here you can improve the loop labeled `.73`. You can replace the five instructions directly after label `.73` with one (`MOVE.W (An)+,(Am)+`) if you place the variable `fz` in an address register instead of in a data register. You can append the three instructions after label `.71` together

using a DBRA instruction. If you later decide to remove this function, then you can place it at .71 with improvements.

Now you will need to add functions to the test.c module, so you can delete lines. It would also be interesting to know how the memory of the individual memory blocks is distributed. Look at the following functions:

```
void print_memory()
{
  struct Memoryblock *blk;
  register UBYTE *ptr,*end;
  register struct Zline *z;
  register struct Memorysection *st;
  struct Zline *fz;
  struct Memorysection *fst;

  printf("Memory construction:\n");
  for (blk = actualEditor->block.head; blk->succ; blk = blk-
>succ)
  {
    printf("\nBlock %8lx, Lenght = %ld, Free = %ld\n",
          blk,blk->length,blk->free);

    ptr = (UBYTE *)(blk + 1);
    end = ptr + blk->length;
    while (ptr < end)
    {
      fz = NULL;
      fst = NULL;

      /* search for the line that is next to the ptr: */
      for (z = actualEditor->zlines.head; z->succ; z = z->succ)
        if ((z >= ptr) && (z < end))
          if (z < ((fz)? fz : (struct Zline *)end))
            fz = z;

      /* No line found or line > prt -> search memory section */
      if ((fz > ptr) || !fz)
      {
        for (st = blk->freeliste.head; st->succ; st = st->succ)
          if ((st >= ptr) && (st < end))
            if (st < ((fst)? fst : (struct Memorysection *)end))
              fst = st;

        /* In case memory section > ptr -> error!!!!! */
        if ((fst > ptr) || !fst)
        {
          printf(" Unused memory section!\n");

          if (fz)
            if (fst)
              if (fst < fz)
                ptr = (UBYTE *)fst;
              else
                ptr = (UBYTE *)fz;
            else
              ptr = (UBYTE *)fz;
          else
            if (fst)
```

```
                ptr = (UBYTE *)fst;
              else
                break;
            }
            else
            {
              /* Memorysection: */
              printf("Memory section (%d)\n",fst->len);

              ptr += sizeof(struct Memorysection) + fst->len;
            }
          }
          else
          {
            /* Zline */
            printf("Line: %s\n",fz+1);

            ptr += sizeof(struct Zline) + EVENLEN(fz->len);
          }

        }
      }

      printf("\n");
}

void delete_line()
{
  register struct Zline *z;
  UWORD nr;

  printf("Number of line that should be erased: [0..n] ");
  scanf("%d",&nr);
  while (getchar() != 10) ;    /* read one CR! */

  for (z = actualEditor->zlines.head; z->succ && nr; z = z-
>succ, nr--);

  if (z->succ)
  {
    printf("Delete: %s\n",z+1);
    deleteZline(z);
  }
}
```

The `print_memory` function looks complicated, but all it does is display the lines and memory sections in the order in which they appear in the block. When a function determines that the block contains a piece of memory that is neither a line nor memory section, an error message is displayed. The function tries to trap the error that the pointer `ptr` sets at the beginning of the next line or memory piece. The pointer `ptr` points to the start of the memory area that is searched, and `end` points to the end of the same.

The second function `erase_line` looks much simpler. This requires a number to be input and deletes the line that corrseponds to the number, starting the count with zero. Unfortunately the `while` loop with `getchar` must exist, because otherwise an end of line remains. This makes sure that you can enter input after the <Return> key is

pressed. There could be a more elegant solution, but it is only a test program designed for functionality. In addition, the test functions will not be in the finished program.

The `switch case` of the `Test` function needs the following lines so that you can examine the new `test` functions:

```
case 's':
   print_memory();
   break;
case 'd':
   delete_line();
   break;
```

Compile the program and examine the individual modules. Add new lines, delete others and constantly examine the memory arrangement. Maybe you will find some things that can be improved. The complete source text and the compiled editor can be found in directory `V0.2` of the optional disk for this book.

4.3.6 Text Output in a Window

This section implements functions that let you display text in the editor's own window instead of the CLI window. As in almost everything, the Amiga and its operating system offer different ways of reaching the same goal.

First you must open the window, then specify the window's size and other qualities. Chapter 3 described window parameters. You have a number of these parameters available.

The output window currently in use is a perfectly normal window. Look at the data in the `NewWindow` structure of the main program, which includes the following numbers:

```
100,40,440,156
```

This is the starting position of the window. We will change this as soon as we have most of the editor functions tested and running.

The following line ensures that your window appears in the same colors as the Workbench window:

```
AUTOFRONTPEN,AUTOBACKPEN
```

The `IDCMP` flags follow next:

```
REFRESHWINDOW | MOUSEBUTTONS | RAWKEY | CLOSEWINDOW | NEWSIZ,
```

The program needs information about redisplaying window data. This information states whether a mouse button or key has been pressed, whether the user has clicked on the window's close gadget, and whether the window's size has changed (more on window size in a moment).

Next you should assign the window flags:

```
WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH | WINDOWCLOSE|
SIZEBOTTOM   | SIMPLE_REFRESH | ACTIVATE,
```

Window sizing

The window should have facilities for enlarging, reduction and movement. You don't want to exit with the close gadget, otherwise the `CLOSEWINDOW` flag would be meaningless. The size gadget for enlarging or reducing appears in the lower right corner of the window. The `ACTIVATE` flag ensures that the window is active at the same time it is opened.

The `SIMPLE_REFRESH` flag is one we haven't mentioned until now. This flag determines the operating system's next task if the user moves another window over the editor window. This action wipes out the contents of the window hidden by the overlapping window. When you remove the overlapping window, the operating system must refresh (redraw) the deleted contents of the window. Chapter 3 mentioned three ways of screen refreshing:

`SIMPLE_REFRESH`	Redraws the window.
`SMART_REFRESH`	Recreates the window from window data available in RAM.
`SUPER_BITMAP`	Stores text and graphic contents of window in a buffer for later recall.

`SIMPLE_REFRESH` or `SMART_REFRESH` works quite well for any window used for simple text display. Since the editor window will stay at the front most of the time (i.e., won't be covered too often), `SIMPLE_REFRESH` is enough in most cases. In addition, the `SIMPLE_REFRESH` window uses less memory than the `SMART_REFRESH` window.

The `SIMPLE_REFRESH` window has one disadvantage that you'll see when you start scrolling. The `ScrollRaster` function moves the screen contents, thus scrolling the window. If you scroll in a `SIMPLE_REFRESH` window with another window covering it, some problem occurs with refreshing the formerly covered section of the window. If you wish, replace `SIMPLE_REFRESH` with `SMART_REFRESH`. `SMART_REFRESH` moves faster, but also uses more memory.

Clipping

There is another problem with text output in your window. You do not want to write in the border of your window. The `GIMMEZEROZERO`

flag provides you with your own RastPort for input inside of the window. The window border has its own RastPort as well, so that you cannot write in the border. But this also requires more memory, and the output is slower because you must manage two RastPorts from Intuition. You want the simplest, more efficient way, so clipping makes sure that your text does not run into the screen border.

Two functions exist for the actual text output. One comes from the `graphics.library` and is called `Text`; the second comes from Intuition and is called `PrintIText`. You assign `Text` a pointer to the RastPort, the text and the length. `PrintIText` needs one pointer to the RastPort, one pointer to the `IntuiText` structure and X and Y coordinates. You can specify the text's writing mode, color, position and the font parameters. The text to which the `IntuiText` structure points must end with a null byte.

At first glance the Intuition functions appear to be more elegant than the graphic functions, which look like sloppy programming. The elegance of Intuition functions come with a price—time. Which special functions do we need to output text in our editor? We need to determine the output position (i.e., `Move` from `graphics.library`). We want to display control codes in red and change the text color back (`SetAPen` from `graphics.library`). We also want reserved C words to appear in bold print (`SetSoftStyle`). Those are all the functions we need from `graphics.library`. We chose the more primitive functions because they execute faster than normal operating system functions.

How can we make sure that text doesn't overwrite the window's border? Simple. We first establish the total size of the window within the borders, measured by character height and width. Then we determine when no more characters should be displayed. We place the width and height of the window in the `Editor` structure, because this value can be shared by multiple windows. Since these two values change as the window size changes, we must reset the window using the `NEWSIZE` flag listed in the `IDCMP` flags. Every time a corresponding report returns, the program recalculates window height and width. A `SIMPLE_REFRESH` window returns a `REFRESHWINDOW` report, and redisplays the contents of the window.

The right and bottom borders still remain. These borders don't belong to either the window border or the text (the window width is not divisible by the letter width). The same goes for the height. The following illustration shows exactly how we solved this problem:

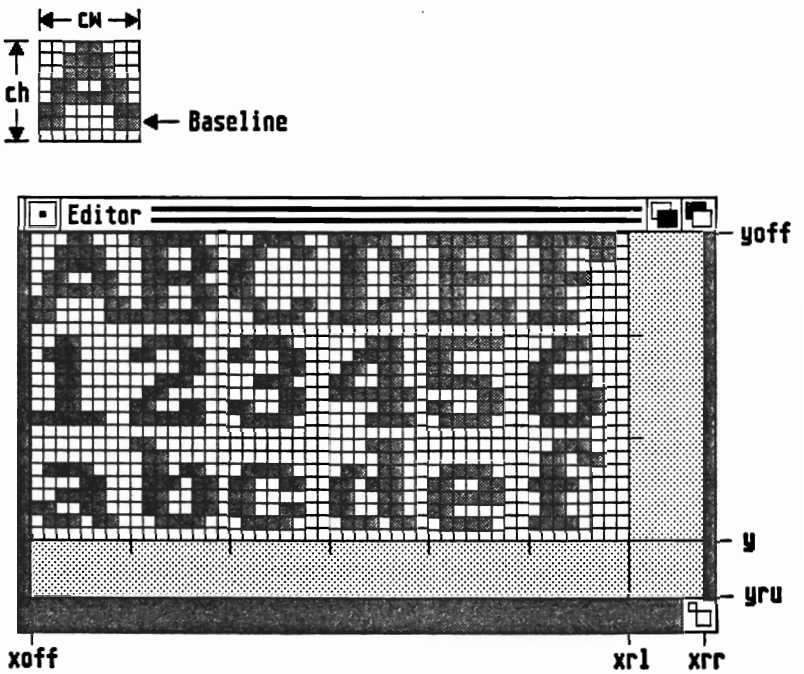


Figure 4.1

The letters are not proportional to make it easier to view. The letter above the window shows an example of the height and width of the characters (cw = character width, ch = character height). Because we don't use any proportional character sets in this example, these values remain the same for all letters. The window width in characters (wcw = WindowCharacterWidth) comes from subtracting the width of the window (window->Width) from the width of the left border (Window->BorderLeft) and right border (window->BorderRight), then dividing the result by the width of one character (cw).

The height is calculated in much the same way:

```
wch = (window->height - window->BorderTop
      - window->BorderBottom) / ch;
```

The width of the left and top screen borders specify the top left position of the window. We save the two values to the Editor structure as xoff (X offset) and yoff (Y offset). The same goes for cw and ch, which can change from window to window if you change the character set using Preferences, then open another window. A pointer to the RastPort of the window belongs in our Editor structure so that we don't always have to access the Window structure. We know that graphic output on the Amiga is through the RastPorts.

You know what to do about the borders. The editor may never need this, but it will keep graphic garbage to a minimum. Each time the program receives a REFRESHWINDOW report it deletes the contents of

both borders. We need additional information for this (e.g., the border positions). The above illustration listed additional coordinates with `xrl` (X, Border, left corner), `xrr` (X, border, right corner), and `yru` (Y, border, bottom corner). We need the top edge of the border, which is given in the drawing by Y. Y provides itself in part, so no calculations are needed yet. Display the text line by line and count the Y coordinates. If either the window fills up or no more text is available, the Y coordinate of the line appears directly beneath the last text line. This corresponds exactly to the top corner of the border.

There are more details given in the program. When positioning the text, be careful that the text always appears relative to its baseline (see illustration above). We must increase the Y position to match the distance of the baseline from the top edge of the letter before calling `Move`.

Now combine the individual functions that handle text output into one function:

```
void printAll()
```

This function displays the entire text. Next the program tests the top line in the window as long as the line appears, until the window fills or the end of the text occurs. The `printLine` function passes control to a pointer to the line and the current Y position. Then the two borders are deleted, provided they are greater than zero.

The next function displays the line left-justified at the Y position:

```
void printLine(line,y)
```

This is not as easy as it sounds. Our editor works with true tabs, which would appear as little graphic boxes. When we let the output run from the console.device, the tabs are displayed correctly. The console.device uses too much memory overhead, which makes the output slower and won't let us start anything. We must convert the tabs ourselves. Another function performs this task:

```
void convertLineForPrint(text,length,width,buffer)
```

This function expects a pointer to the text (not the `Line` structure) and its length. You must assign a pointer to a buffer in which the converted lines are written, because spaces fill in the buffer to the end of the line. This function doesn't display a visible CR/LF, because the CR or LF should not appear at the end of a line (for aesthetic reasons, if for nothing else). If the line doesn't end with a CR, LF or CR/LF, the function fills the rest of the line with dots (`CHR$(183)`). This also applies to lines that don't completely fit in the window because they are too long. A point at the end of a line means that the lines continues, and multiple points mean that the line doesn't conclude with a normal end of line character.

We can now convert and display the line using the `Text` graphic function. Because a line can include spaces, or because the line is narrower than the window, the line could be indented for formatting reasons. This formatting becomes inefficient. Displaying spaces requires the same amount of time as displaying a letter or number. A faster method would be to replace any larger number of spaces by drawing a rectangle the same size as the spaces to be displayed. The following function performs this task:

```
void printAt(buffer,length,x,y)
```

The `printAt` function also handles control character output. Control codes are displayed in red. For example, `CHR$(1)` appears as a red "A" (`"A" = CHR$(1 | 64)`. A pointer to the text controls length and the X coordinate position according to `xoff`.

One remaining text display function controls output, `wcw`, `wch`, `xoff`, etc.:

```
void initWindowSize(ed)
```

The variable `ed` points to the `Editor` structure whose value should be tested. Before we look at these functions, we should review our list of the features that our editor should have.

Folding

Folding refers to the ability to fold text into areas in memory where the line wouldn't normally be visible. Usually you would search through hundreds of lines to find a routine. Folding allows you to move routines into their own areas, marked by the following string:

```
/*#FOLD: Routine_Name */
```

Each `#FOLD` statement acts as a line pointer which indicates a set of folded lines behind the statement. The pointer must be activated to get to the folded segment. This takes less time than scrolling through a full listing. Because the maximum number of lines that fits in the screen decreases, 50 (same as 400, the maximum resolution) divided by 8 (character height), we can also determine the size of this field.

We first want to expand the include file `Editor.h`:

```
#define MAXHEIGHT 50
#define FGPEN 1L
#define BGPEN 0L
#define CTRLPEN 3L
#define CONTROLCODE(c) (((c)&127)<32)

struct Editor
{
  struct Editor *succ;
  struct Editor *pred;
  struct Window *window;
  struct BList block;
  struct ZList zlines;
  UBYTE buffer[MAXWIDTH];
```

```
  UBYTE tabstring[MAXWIDTH];
  UWORD num_lines;
  struct Zline *actual;
  struct Zline *zlinesptr[MAXHEIGHT];
  UWORD toppos;
  UWORD xpos,ypos;
  UWORD changed:1;
  UWORD insert:1;
  struct RastPort *rp;
  UWORD xoff,yoff;
  UWORD cw,ch;
  UWORD wcw,wch;
  UWORD xrl,xrr,yru,bl;
};
```

In addition to the maximum window height, the standard foreground color (FGPEN = ForegroundPen), the background color (BGPEN = BackgroundPen) and the color for the control codes (CTRLPEN) are defined. We also define what a control code is (CONTROLCODE)—a character with a code less than 32, or greater than or equal to 128 and less than 160. The Amiga ASCII table displays these characters as rectangles. If you use your own character set in which you have assigned your own character to these codes, you can change the definition of CONTROLCODE.

The `editor` structure changes slightly. Instead of the top pointer there is now the `zlinesptr` array. `zlinesptr[0]` serves the same function as `top` had before. Additional elements for text output are added to the end of the structure as discussed previously. The two functions of the `Memory.c` module that make use of `actual` must be rewritten. In the function `garbageCollection` the lines where `zprt` appears must be defined as `struct line **zptr`. Search for the following lines:

```
if (actualEditor->actual == z)
   actualEditor->actual = fz;
```

And add the following lines:

```
for (n = 0, zptr = actualEditor->zlinesptr;
  n <= actualEditor->wch; n++, zptr++)
  if (*zptr == z)
  {
    *zptr = fz;
    break;
  }
```

The function `deleteZline` also needs a change. The lines listed below:

```
if (actualEditor->actual == zline)
   actualEditor->actual = NULL;
```

require the addition of `zprt` and `n (UWORD)`. Add to the above lines the following:

```
for (n = 0, zptr = actualEditor->zlinesptr;
     n <= actualEditor->wch; n++, zptr++)
   if (*zptr == line)
   {
      *zptr = NULL;
      break;
   }
```

Now we come to the changes in the main `Editor.c` module. `printAll` is not the only function that we are going to add:

```
void initWindowSize(),SetDrMd(),SetAPen(),SetBPen();
void printAll(), BeginRefresh(),EndRefresh();
```

The following two new functions test for one line before or after the current line. These may seem unnecessary, but the folding extension will need them later:

```
/*****************************************
 *
 * nextLine(line)
 *
 * Returns pointer to next line
 * (and executes folding)
 * zero if not next line
 *
 * line ^ line structure.
 *
 *****************************************/

struct Zline *nextLine(line)
register struct Zline *line;
{
  register struct Zline *z;

  if (line)
  {
    if (z = line->succ)
      if (!(z->succ))
        z = NULL;

    return (z);
  }
  else
    return (NULL);
}

/*****************************************
 *
 * prevLine(line)
 *
 * Returns pointer to previous line
 * (and executes folding)
 * zero of no previous line
 *
 *
 * line ^ line structure
 *
 *
 *****************************************/
```

```
struct Zline *prevLine(line)
register struct Zline *line;
{
  register struct Zline *z;

  if (line)
  {
    if (z = line->pred)
      if (!(z->pred))
        z = NULL;

    return (z);
  }
  else
    return (NULL);
}
```

The `OpenEditor` function was also expanded:

```
struct Editor *OpenEditor()
{
  register struct Editor *ed = NULL;
  register struct Window *wd;
  register struct RastPort *rp;
  register ULONG flags;
  register UBYTE *ptr;
  register struct Zline **zptr;
  register UWORD n;

  /* IDCMPFlags saved, sets it to zero in structure
    => use your own UserPort! */
  flags = newEdWindow.IDCMPFlags;
  newEdWindow.IDCMPFlags = NULL;

  /* Allocate memory for memory structure: */
  if (ed = malloc(sizeof(struct Editor)))
    /* Window open: */
    if (wd = OpenWindow(&newEdWindow))
    {
      ed->window = wd;
      ed->rp = (rp = wd->RPort);

      /* Set write mode: */
      SetDrMd(rp,JAM2);
      SetAPen(rp,FGPEN);
      SetBPen(rp,BGPEN);

      /* UserPort established: */
      wd->UserPort = edUserPort;
      ModifyIDCMP(wd,flags);

      /* Parameter initialization: */
      NewList(&(ed->block));
      NewList(&(ed->zlines));
      ed->num_lines = 0;
      ed->actual = NULL;
      ed->toppos = 0;
      ed->xpos = 1;
      ed->ypos = 1;
      ed->changed = 0;
      ed->insert = 1;
```

```
      /* zlinesptr-Array initialization: */
      for (n = 0, zptr = ed->zlinesptr; n < MAXHEIGHT; n++,
zptr++)
        *zptr = NULL;

      /* Tab initialization: */
      ptr = ed->tabstring;
      *ptr++ = 1;
      for (n = 1; n < MAXWIDTH; n++)
        if (n % 3)
          *ptr++ = 1;
        else
          *ptr++ = 0;

      ed->wch = 0;
      initWindowSize(ed);
    }
    else
    {
      free(ed);
      ed = NULL;
    }

  newEdWindow.IDCMPFlags = flags;
  return (ed);
}
```

Tab initialization gives the user a couple of predefined tabs. Two new flags must be inserted in the routine which checks the report types in the main program:

```
        case NEWSIZE:
        initWindowSize(actualEditor);
print();
          break;

        case REFRESHWINDOW:
        BeginRefresh(actualEditor->window);
          printAll();
        EndRefresh(actualEditor->window,TRUE);
          break;
```

The functions `BeginRefresh` and `EndRefresh` should be called after a `REFRESHWINDOW` event so Intuition knows that the window was restored. The functions also ensure that only the sections of the screen that need it have been renewed (see Chapter 2 for information). The `print` function originated in the `test.c` module. Before we assign this, let's implement a new module named `output.c`:

```
<src/Output.c>

/************
 *
 * Includes:
 *
 ************/

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

```
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

#define CR 13
#define LF 10
#define TAB 9

/**********************
 *
 * External Functions:
 *
 **********************/

void SetAPen(),RectFill(),Text(),Move();
struct Zline *nextLine();

/*********************
 *
 * External Variables:
 *
 *********************/

extern struct Editor *actualEditor;

/***************
 *
 * Functions: *
 *
 ***************/

/*********************************
 *
 * initWindowSize(ed)
 *
 * Initializes the width/height
 * statement of the actual windows.
 * Restores the zlineptr field
 *   .
 *
 * ed ^ Editor structure.
 *
 *********************************/

void initWindowSize    (ed)
register struct Editor *ed;
{
  register struct Window *w;
  register struct RastPort *rp;
  register struct Zline *z,**zptr;
  register UWORD n,newh;

  w   = ed->window;
  rp = ed->rp;

  ed->xoff = (UWORD)w->BorderLeft;
  ed->yoff = (UWORD)w->BorderTop;
```

```
  ed->cw   = (rp->TxWidth)? rp->TxWidth : 8;
  ed->ch   = (rp->TxHeight)? rp->TxHeight : 8;

  ed->wcw = (w->Width - ed->xoff - w->BorderRight) / ed->cw;
  newh   = (w->Height- ed->yoff - w->BorderBottom)/ ed->ch;

  ed->xrl = ed->xoff + ed->wcw*ed->cw;
  ed->xrr = w->Width - w->BorderRight - 1;
  ed->yru = w->Height - w->BorderBottom - 1;
  ed->bl   = rp->TxBaseline;

  /* Now restore the zlineptr field: */
  if (newh != ed->wch)
  {
    if (newh < ed->wch)
    {
      /* Pointer to Null set: */
      zptr = ed->zlinesptr + newh + 1;
      for (n = newh; n < ed->wch; n++)
        *zptr++ = NULL;
    }
    else
    {
      zptr   = ed->zlinesptr + ed->wch;
      z      = *zptr++;
      for (n = ed->wch; n < newh; n++)
        *zptr++ = (z = nextLine(z));
    }

    ed->wch = newh;
  }
}

/*************************************
 *
 * convertLineForPrint(line,len,buf)
 *
 * Convert line so that this can be
 * displayed simply.
 * Tabs converted into spaces
 *
 *
 * line ^ Zline.
 * len = the length.
 * w = maximum wiidth.
 * buf ^buffer, in which the converted line
 *       is stored.
 *
 *************************************/

void convertLineForPrint(line,len,w,buf)
UBYTE      *line;
register WORD     len;
register UWORD     w;
register UBYTE   *buf;
{
  register UBYTE *tab;
  register UWORD l = 1;
  register UBYTE lastchar;

  /* determine if ends with CR or LF: */
  if (len)
```

```
  {
    tab = line + len - 1;
    if (*tab == CR)
    {
      len--;
      lastchar = ' ';
    }
    else if (*tab == LF)
    {
      len--;
      lastchar = ' ';
      if ((len) && (*--tab == CR))
        len--;
    }
    else
      lastchar = 183;
  }
  else
    lastchar = 183;

  /* Zline convert: */
  tab = actualEditor->tabstring;
  while ((len--) && (l < w))
    if ((*buf = *line++) == TAB)
      do
      {
        *buf++ = ' ';
        l++;
        tab++;
      } while ((*tab) && (l < w));
    else
    {
      buf++;
      tab++;
      l++;
    }

  /* determine if the line is not completely converted: */
  if (len >= 0)
    lastchar = 183;

  /* line filled from last character to end: */
  while (l++ <= w)
    *buf++ = lastchar;
}

/***********************************
 *
 * printAt(buf,len,x,y)
 *
 *Prive character string buf a position
 * (x,y) in a window. The character-
 * string is changed!
 *
 * buf ^ character sring.
 * len = the length.
 * x = Pixel-X-Position.
 * y = Pixel-Y-Position.
 *
 ***********************************/

void printAt   (buf,len,x,y)
```

```
register UBYTE  *buf;
WORD        len;
register UWORD  x,y;
{
  struct RastPort *rp = actualEditor->rp;
  register UWORD cw = actualEditor->cw;
  register UWORD x2;
  UWORD y2 = y + actualEditor->ch - 1;
  UBYTE *ptr;
  ULONG l;
  while (len > 0)
    if (*buf == ' ')
    {
      /* output spaces: */
      x2 = x - 1;

      while ((*buf == ' ') && len--)
      {
        buf++;
        x2 += cw;
      }

      SetAPen(rp,BGPEN);
      RectFill(rp,(ULONG)x,(ULONG)y,(ULONG)x2,(ULONG)y2);
      SetAPen(rp,FGPEN);

      x = ++x2;
    }
    else
    {
      /* Also output text: */
      Move(rp,(ULONG)x,(ULONG)y + actualEditor->bl);
      ptr = buf;

      if (CONTROLCODE(*buf))
      {
        SetAPen(rp,CTRLPEN);

        while (CONTROLCODE(*buf) && len--)
          *buf++ |= 64;
        l = buf - ptr;
        Text(rp,ptr,l);

        SetAPen(rp,FGPEN);
      }
      else
      {
        while (!CONTROLCODE(*buf) && (*buf != ' ') && len--)
          buf++;
        l = buf - ptr;
        Text(rp,ptr,l);
      }
      x += cw*l;
    }
}

/************************************
 *
 * printLine(line,y)
 *
 * Displays Zline at pixel oosition
 *y on the screen.
```

```
 *
 * line ^ Zline structure.
 * y = pixel positon.
 *
 ************************************/

void printLine    (line,y)
register struct Zline *line;
register UWORD    y;
{
  static UBYTE buf[MAXWIDTH];
  register UWORD w;

  convertLineForPrint(line+1,line->len,w = actualEditor-
>wcw,buf);
  printAt(buf,w,actualEditor->xoff,y);
}

/*********************************
 *
 * printAll
 *
 * redisplays the entire screen.
 *
 *********************************/

void printAll()
{
  register UWORD y,ch,count;
  register struct Zline **z;
  register struct RastPort *rp;

  y    = actualEditor->yoff;
  ch   = actualEditor->ch;
  count = actualEditor->wch;
  rp   = actualEditor->rp;

  /* Output the line: */
  for (z = actualEditor->zlinesptr; count-- && (*z != NULL);
    z++, y += ch)
    printLine(*z,y);

  /* erase right and bottom border: */
  SetAPen(rp,BGPEN);
  if (actualEditor->yru >= y)
    RectFill(rp,(ULONG)actualEditor->xoff,(ULONG)y,
    (ULONG)actualEditor->xrr,(ULONG)actualEditor->yru);
  if ((actualEditor->xrr >= actualEditor->xrl)
    && (y > actualEditor->yoff))
    RectFill(rp,(ULONG)actualEditor->xrl,(ULONG)actualEditor-
>yoff,
            (ULONG)actualEditor->xrr,(ULONG)y-1);
  SetAPen(rp,FGPEN);
}
```

Now here are a few comments about the above functions and the special additions:

- The `initWindowSize` function performs a security check with the size of the character set and a null.

- If the window was enlarged or reduced, either additional pointers are calculated or the unnecessary pointers are set to zero. The `nextLine` function will make the implementation of folding easier. Then we only need to change the `nextLine` and `prevLine` functions. All other functions that access these two functions can work with folding.

- `convertLineForPrint` searches for the end of line and sets the variable `lastchar`. This variable contains the characters that fill in the rest of the line. The `tab` variable acts as the pointer to the end of line. Be sure that there is at least one character (usually a space or a period) displayed behind the actual text of the line. If the window is 70 characters wide, the line is only 69 characters wide.

- In `printAt` the `x2` and `y2` variables contain the coordinates of the bottom right corner for the rectangle that must be displayed instead of blank spaces. The background color must be assigned to the rectangle beforehand. Otherwise the program displays a white (visible) rectangle.

- The many type conversions are necessary because we must assign LONG values to the operating system functions.

- The function `printLine` defines the buffer as `static`, otherwise it takes up too much room on the stack. In addition, other local variables have space reserved on the stack, but no more than 80 bytes at a time.

- Calling the `RectFill` function with the coordinates that separate the left corner from the right or the top corner from the bottom make the Amiga run faster. The check in `printAll` is very important to this. Besides the definition of `xrl`, `xrr` and `yru`, the borders may be assigned negative widths. (i.e., if the window's width or height is divisible by the width or height or a character). In that case no border can be displayed.

So far, so good. We need changes to the `test.c` module. First declare a few external functions:

```
struct Zline *nextLine();
void printAll();
```

Now rewrite the `print` function so that it supplies the values to the pointer established in the `zlinesptr` array:

```
void print()
{
  register struct Zline **z;
  register UWORD n;

  for (n = 0, z = actualEditor->zlinesptr; n <= actualEditor-
>wch;
```

```
      n++,z++)
    printf("Line %d an Adresse %6lx.\n",n,*z);
}
```

There is a new function for initializing the `zlinesptr` field. It won't be needed later in the actual program, so we placed them in the `test.c` module:

```
void init_zlinesptr()
{
  register struct Zline *z,**zptr;
  register UWORD n;

  for (n = 0, z = (struct Zline *) &(actualEditor->zlines.head),
    zptr = actualEditor->zlinesptr; n < MAXHEIGHT; n++)
    *zptr++ = (z = nextLine(z));
}
```

Finally the actual test program needs a change. After inserting and erasing lines, `zlinesptr` must be re-initialized. Furthermore, the `printAll` function must be called by the command `p` (print). The corresponding branch of the `switch case` application looks like the following:

```
      case 'a':
        input();
        init_zlinesptr();
        break;
      case 'p':
        print();
        printAll();
        break;
      case 'b':
        print_blocks();
        break;
      case 's':
        print_memory();
        break;
      case 'd':
        delete_line();
        init_zlinesptr();
        break;
```

Next add the following line and object definition (`OBJ=...`) to the `makefile`.

```
src/Output.o: src/Output.c src/Editor.h pre/Editor.pre
```

The complete editor source can be found in the `V0.3` directory of the optional diskette for this book. After compiling start the test editor and enter more lines by pressing `a` (append). Exit the test module by pressing <Esc>. You can move the window, enlarge it, reduce it, etc. Make sure that the window display remains consistent.

Now we'll concentrate on making the editor a more efficient program. You remember that the C compiler compiles the expression `*var++`

incorrectly. Remove the increment (++) and process this separately—the C compiler creates a more efficient object code. Compile the following test program to create its assembler source text:

```
main()
{
   register char *ptr;
   char string[10];
   register short n;

   for (n = 0, ptr = string; n < 10; n++)
      *ptr++ = 0;

   for (n = 0, ptr = string; n < 10; n++, ptr++)
      *ptr = 0;
}
```

Both `for` loops increment the pointer `ptr`. Compile the program using the following assignment:

```
cc test.c -a -t
```

You get the assmebler source text with the name `Test.asm`, which looks like the following using our Aztec C compiler:

```
;                       main()
;                       {
   public _main
_main:
   link a5,#.2
   movem.l .3,-(sp)
;                          register char *ptr;
;                          char string[10];
;                          register short n;
;
;                    for (n = 0, ptr = string; n < 10; n++)
   move.l #0,d4
   lea -10(a5),a0
   move.l a0,a2
   bra .7
.6
;                                *ptr++ = 0;
   move.l a2,a0
   add.l #1,a2
   clr.b (a0)
.4
   add.w #1,d4
.7
   cmp.w #10,d4
   blt .6
.5
;
;           for (n = 0, ptr = string; n < 10; n++, ptr++)
   move.l #0,d4
   lea -10(a5),a0
   move.l a0,a2
   bra .11
.10
;                                *ptr = 0;
   clr.b (a2)
```

```
.8
   add.w #1,d4
   add.l #1,a2
.11
   cmp.w #10,d4
   blt .10
.9
;                          }
.12
   movem.l (sp)+,.3
   unlk a5
   rts
.2 equ -10
.3 reg d4/a2
   public .begin
   dseg
   end
```

Look at the two loop bodies in particular, for the first loop from label.6 to label.7, and for the second loop from label.10 to label.11. You see yourself that the first loop needs four commands while the second has three commands. Therefore it is especially important that these commands be run through as often as the loop counter allows. This command becomes more powerful the more often you execute the loop.

It is worthwhile to examine the assembler source text. There the compiler manufacturer tries to make their product as efficient as possible. The loop counter procedure is discussed later in this book.

4.3.7 The Cursor

The editor should let the user move the cursor to any point in a source text for easy editing. The following section shows how to expand the editor for cursor movement using either cursor keys or mouse. When the cursor reaches the bottom or top border of the window, the window contents must scroll to allow access to the next or previous lines in the window. This vertical cursor movement is important, since almost every source text has more lines than can fit into just one window. Horizontal scrolling must be available as well. When the user reduces the size of the editor window, lines that are wider than the reduced window must also be accessible through scrolling.

Folding

We must consider folding when dealing with cursor movement. Remember when folding, two lines that lie directly under each other in the editor window do not have to follow directly after one another in the text. When a line contains a fold starting mark, all lines following this mark, up to the corresponding fold end mark are invisible. The next visible line in the window is the one following the fold end mark. As

with the `zlinesptr` array, an array is needed that contains the line number for every field that is visible in the editor window.

The editor specifies the cursor's line position through the `zlinesptr` array. In addition to the true line position, the `zlinesnr` array needs the window position. This also allows testing without the implementation of folding, before including the folding routine.

Tabs

Tabs present the next problem. The editor Z (included with the Aztec C compiler) forces the cursor to jump over multiple columns if it lands on a tab. Cursor movement depends on the character on which the cursor is found. This is why the cursor stands on the last character of a line instead of behind a line. The cursor position should be fully selectable, even though it causes a few problems with character input (e.g., what happens if you overwrite a tab with another character).

Scrolling

Horizontal scrolling is slightly different from vertical scrolling. When the contents of a window scrolls to the right, the `printAll` function must redisplay the left border. Then a problem arises if the last character of the output is either a space or a dot. This creates a line which is wider than the window (see the `lastchar` variable in the `ConvertLineForPrint` function). That means that when we scroll only one column, dots appear in the first column. We must confess that we noticed this problem when we first examined this. We got around it by converting one more column than was actually needed, and omitting this last character when displaying the line.

Another problem occurs when redisplaying the right border if the last character is a space or a dot. When we move the window contents to the right, the character in the last column must be overwritten.

The new right border output occurs when we scroll to the left. When accessing `printAll`, the editor must be instructed to view the window with a wide left border. This displays the text at the right border. We must display one more column than was scrolled because the last column contains only spaces and dots.

A number of functions could be created that would solve the problem. However, when creating functions for fancy output, it's easy to lose the perspective of exactly what is affected from the changes. We avoided making the output functions too complicated, and leave you to make any modifications to the scrolling that you might want.

Now we return to folding. A fold starting mark defines the beginning of a fold. A typical starting mark, which appears in the current level of text, looks like this:

```
/*#FOLD: */
```

The fold ending mark appears at the end of the text the programmer wants folded, and looks like this:

```
/*#ENDFD*/
```

Both marker strings must appear at the beginning of a line (no leading spaces). You may have noticed that the marks begin with the two characters that normally start comments in C (`/*`). This is necessary because the C compiler must skip these lines. Later we'll show you how to change the marks so that the programmer can use the folding technique in other languages.

The words `#FOLD` and `#ENDFD` must be entered in capital letters. The editor will not fold lines using lower case lettering (e.g., `#Fold` or `#endfd` are unacceptable to the editor).

The colon following the word `#FOLD` lets you add comments or function names. This can tell the user the purpose of the program text within the fold. A completed fold displays the line as seen above and the commentary. For example:

```
/*#FOLD: InnerRoutine */
```

Fold levels

To read a fold, move the cursor onto the starting mark. Press <Ctrl><f> (f as in "fold"). The editor displays the contents of the fold. You can have multiple levels of folding (inserting folds within folds). When creating folds, the editor defines the level of each set of lines and the depth of the fold in which the lines lie. The first line in a text lies at level zero. A fold starting mark increments the level by one, and a fold ending mark decrements the level by one.

The following example displays the fold level on the left margin. The lines on the right correspond to the lines within the fold level:

```
0   Line_1
0   /*#FOLD: Line_2 */
1      Line_3
1      /*#FOLD: Line_4 */
2         Line_5
2         /*#ENDFD Line_6*/
1      Line_7
1      /*#ENDFD Line_8 */
0   Line_9
0   /*#FOLD: Line_10 */
1      /*#FOLD: Line_11 */
2         Line_12
2         /*#ENDFD Line_13 */
1      Line_14
1      /*#ENDFD Line_15 */
0   Line_16
```

The `Editor` structure defines two variables named `minfold` and `maxfold`. Only the lines whose levels lie between `minfold` and `maxfold` are displayed in the window.

<Ctrl><f> enters the next fold level in two ways:

- When the cursor lies at a fold starting mark, `minfold` and `maxfold` are set to the level of the following line so that you see only the corresponding fold.

- Otherwise only `maxfold` increases by one so that you can only see the fold that lies one level higher, in addition to the fold currently in view.

You can exit the current fold by pressing <Ctrl><e> (e as in "exit"). You cannot exit a fold using the cursor keys; the cursor stays inside of the fold currently being accessed.

<Ctrl><e> decreases `maxfold` by one. When `minfold` is greater than `maxfold`, it changes to the same value as `maxfold`. The example text mentioned above can be accessed by the following process:

- `minfold` and `maxfold` are initially equal to zero so that the text from its top level looks like this:

```
0  Line_1
0  /*#FOLD: Line_2 */
0  Line_9
0  /*#FOLD: Line_10 */
0  Line_16
```

- You see only the lines whose fold level is between `minfold` and `maxfold`, or equal to zero. Press <Ctrl><f> while the cursor is in `line_1`. Most of the text unfolds, revealing the following:

```
0  Line_1
0  /*#FOLD: Line_2 */
1     Line_3
1     /*#FOLD: Line_4 */
1     Line_7
1     /*#ENDFD Line_8 */
0  Line_9
0  /*#FOLD: Line_10 */
1     /*#FOLD: Line_11 */
1     Line_14
1     /*#ENDFD Line_15 */
0  Line_16
```

- Pressing <Ctrl><f> again while the cursor is in `line_1` unfolds the entire text, since `maxfold` becomes equal to 2. Press <Ctrl><e> twice to display the lines with a fold level of zero. Move the cursor to `line_2` and press <Ctrl><f>. The following fold contents appear:

```
1     Line_3
1     /*#FOLD: Line_4 */
1     Line_7
1     /*#ENDFD Line_8 */
```

This is confusing. Shouldn't folding display the current fold, instead of these extra folds? When you set `maxfold` to the maximum line level while `minfold` stays at zero, you'll see fold marks in the text even if they weren't selected. This is why the program displays all of the current fold's start and end marks in black characters. When you come across a fold start mark in white characters, this means that you can access the fold.

Implememting We need four functions to move the cursor in the proper directions:

```
cursorRight, cursorLeft, cursorUp, cursorDown
```

These need no arguments, and return either TRUE (the cursor moves) or FALSE (the cursor remains stationary, or reaches the end of a line). We have no use for this return value at the moment. However, if you wish to add more capabilities to the editor later on, knowing whether or not the cursor actually moved can be useful. This knowledge can be used in conjunction with interrupt criteria for loops.

These cursor movement functions call other functions that scroll the editor window contents in the corresponding direction:

```
scrollLeft, scrollRight, scrollDown, scrollUp
```

Cursor movement functions must call opposing scrolling functions. For example, `cursorRight` calls `scrollLeft`. We assign the number of lines or columns that should be scrolled as an argument. The argument type is `UWORD`. All scrolling functions are of type `void` (they return no result).

The next function receives the key presses and accesses the corresponding cursor functions. This function must also handle keyboard input and output:

```
void handleKeys(buf,len)
UBYTE          *buf;
UWORD               len;
```

Cursor display Cursor movement is useless without displaying a visible cursor. The `Cursor` function displays the cursor in `COMPLEMENT` mode:

```
void Cursor()
```

When we call this function before the `Wait` function in the main loop, the cursor appears when the editor is inactive. When the program is busy, the cursor disappears and cannot be used.

Now that you know the important functions for the new module, add these items to the `Editor.h` file:

```
#define FOLDPEN 2L

#define ZLINESPTR(n) aktuellerEditor->zeilenptr[n]
#define ZLINESNR(n) aktuellerEditor->zeilennr[n]
```

```
#define ZLF_FSE 64
#define ZLF_FOLD 63
```

FOLDPEN is the color in which the fold marks are displayed (usually black). The ZLINESPTR and zLINESNR functions should ease access to the elements of corresponding fields. Instead of:

```
actualEditor->linesptr[n]
```

write:

```
ZLINESPTR(n)
```

The last two defines belong to the line structure flags in the data structure `Line`. When the flag `ZLF_FSE` is set, it handles the line as a fold mark. It doesn't matter whether it is handled as a fold start or fold end mark (FSE = Fold Start End). `ZLF_FOLD` specifies the fold level of the line (for non-folded lines `ZLF_FOLD`=0). Be careful that the fold start marks themselves don't belong to the fold. The maximum fold level is 63.

The expanded `Editor` structure looks like this:

```
struct Editor
{
  struct Editor *succ;
  struct Editor *pred;
  struct Window *window;
  struct BList block;
  struct ZList zlines;
  UBYTE buffer[MAXWIDTH];
  UBYTE tabstring[MAXWIDTH];
  UWORD num_lines;
  struct Zline *actual;
  struct Zline *zlinesptr[MAXHEIGHT];
  UWORD zlinesnr[MAXHEIGHT];
  UWORD toppos,leftpos;
  UWORD xpos,ypos;
  UWORD wdy;
  UWORD changed:1;
  UWORD insert:1;
  struct RastPort *rp;
  UWORD xoff,yoff;
  UWORD xscr,yscr;
  UWORD cw,ch;
  UWORD wcw,wch;
  UWORD xrl,xrr,yru,bl;
  UWORD minfold;
  UWORD maxfold;
};
```

The following elements are new:

```
zlinesnr[]
```

This contains the numbers of all of the lines that are visible in the window (see also `zlinesptr`).

`leftpos`

When the window scrolls to the left, the first column of the window changes to the next column up. `leftpos` specifies the number of the column directly to the left of the visible window. `leftpos` has a default value of zero, which means that the first visible column in the window has the number one.

`wdy`

This specifies the line position of the cursor with a value between zero and (`actualEditor->wch-1`).

`xscr, yscr`

These variables supply the right bottom border of the window (necessary for scrolling). For text output `xscr = (xr1 - 1)` and `yscr = (y - 1)` (measured in pixels).

`minfold, maxfold`

Specifies the minimum and maximum fold levels that determine which folds appear on the screen and which don't.

The following listing for the `Cursor` module contains all of the previously described cursor functions:

```
<src/Cursor.c>

/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

#define CSI 0x9B
#define CUU 'A'
#define CUD 'B'
#define CUF 'C'
#define CUB 'D'
#define SU 'S'
#define SD 'T'

#define CFOLD 6
#define CENDF 5

/**********************
 *
```

```
 * External Functions:
 *
 *********************/

void SetDrMd(),RectFill(),printAll();
struct Zline *nextLine(),*prevLine();

/*********************
 *
 * External Variables:
 *
 *********************/

extern struct Editor *actualEditor;
extern UWORD SplitDec;

/***************
 *           *
 * Functions: *
 *           *
 ***************/

/************************************
 *
 * restoreZlinenptr:
 *
 * Restore the zlinesptr/nr field
 * for all lines in window plus
 * the next line.
 * The first line must be
 * correct!
 *
 ************************************/

void restoreZlinenptr()
{
  register UWORD n;
  register struct Zline **zptr,*z;
  register UWORD *pnr;
  UWORD znr;

  for (n = actualEditor->wch, zptr = actualEditor->zlinesptr,
          pnr = actualEditor->zlinesnr,
          z = *zptr++, znr = *pnr++; n; n--)
  {
    *zptr++ = (z = nextLine(z,&znr));
    *pnr++  = znr;
  }
}

/*********************************
 *
 * Cursor:
 *
 * Set or erase the cursor
 * at the actual position.
 *
 *********************************/

void Cursor()
{
  register struct RastPort *rp;
```

```
  register LONG x,y;
  register LONG cw,ch;

  SetDrMd(rp = actualEditor->rp,COMPLEMENT);

  x = (actualEditor->xpos - actualEditor->leftpos - 1)
    *(cw = actualEditor->cw) + actualEditor->xoff;
  y = actualEditor->wdy
    *(ch = actualEditor->ch) + actualEditor->yoff;

  RectFill(rp,x,y,x + cw-1,y + ch-1);

  SetDrMd(rp,JAM2);
}

/*******************************
 *
 * scrollRight(num):
 *
 * Scroll the window contents
 * num characters to the right.
 *
 * num = number of characters.
 *
 *******************************/

void scrollRight(num)
register UWORD    num;
{
  register UWORD y,ch,count;
  register struct Zline **z;
  UWORD wcw,xoff,leftpos;

  actualEditor->leftpos -= num;

  if (num >= actualEditor->wcw)
    printAll();
  else
  {
    /* Scrolling: */
    ScrollRaster(actualEditor->rp,
             -num * (LONG)actualEditor->cw,0L,
             (LONG)actualEditor->xoff,(LONG)actualEditor->yoff,
             (LONG)actualEditor->xscr,(LONG)actualEditor->yscr);

    /* now list printAll(): */
    wcw = actualEditor->wcw;
    actualEditor->wcw = num + 1;
    SplitDec = 1;

    y     = actualEditor->yoff;
    ch    = actualEditor->ch;
    count = actualEditor->wch;

    /* Output line: */
    for (z = actualEditor->zlinesptr; count-- && (*z != NULL);
        z++, y += ch)
      printLine(*z,y);

    SplitDec = 0;

    /* Display right border beside last character */
```

```
    leftpos = actualEditor->leftpos;
    xoff    = actualEditor->xoff;
    actualEditor->wcw      = 2;
    actualEditor->leftpos += (y = wcw - 2);
    actualEditor->xoff    += actualEditor->cw * y;

    y     = actualEditor->yoff;
    count = actualEditor->wch;

    /* Output line: */
    for (z = actualEditor->zlinesptr; count-- && (*z != NULL);
        z++, y += ch)
      printLine(*z,y);

    actualEditor->wcw    = wcw;
    actualEditor->leftpos= leftpos;
    actualEditor->xoff   = xoff;
  }
}

/*******************************
 *
 * scrollLeft(num):
 *
 * Scroll the window contents
 * num characters left.
 *
 * num = number of characters.
 *
 *******************************/

void scrollLeft(num)
register UWORD  num;
{
  register UWORD y,ch,count;
  register struct Zline **z;
  UWORD wcw,xoff,leftpos;

  actualEditor->leftpos += num;

  if (num >= actualEditor->wcw)
    printAll();
  else
  {
    /* Scrolling: */
    ScrollRaster(actualEditor->rp,
             num * (LONG)actualEditor->cw,0L,
             (LONG)actualEditor->xoff,(LONG)actualEditor->yoff,
             (LONG)actualEditor->xscr,(LONG)actualEditor->yscr);

    /* now list printAll(): */
    wcw    = actualEditor->wcw;
    leftpos = actualEditor->leftpos;
    xoff    = actualEditor->xoff;
    actualEditor->wcw      = num + 1;
    actualEditor->leftpos += (y = wcw - num - 1);
    actualEditor->xoff    += actualEditor->cw * y;

    y   = actualEditor->yoff;
    ch    = actualEditor->ch;
    count = actualEditor->wch;
```

```
    /* Output line: */
    for (z = actualEditor->zlinesptr; count-- && (*z != NULL);
        z++, y += ch)
      printLine(*z,y);

    actualEditor->wcw    = wcw;
    actualEditor->leftpos= leftpos;
    actualEditor->xoff   = xoff;
  }
}

/*****************************
 *
 * scrollUp(num):
 *
 * Scroll the window contents
 * num lines up.
 * Num lines must
 * exist!
 *
 * num = number of lines.
 *
 *****************************/

void scrollUp (num)
register UWORD num;
{
  register struct Zline *z,**zptr,**zold;
  register UWORD n,wch = actualEditor->wch,y;
  UWORD znr,*pnr,*oldnr;

  if (num >= wch)
  {
    /* Display window without Scrolling */
    /* First look for top line: */
    n = wch;
    if (num > wch)
    {
      z = ZLINESPTR(n);
      znr = ZLINESNR(n);
      while (++n < num)
        z = nextLine(z,&znr);
    }
    else
    {
      z = ZLINESPTR(n - 1);
      znr = ZLINESNR(n - 1);
    }

    /* z points before the top line of the window */
    for (n = 0, zptr = actualEditor->zlinesptr,
        pnr = actualEditor->zlinesnr; n <= wch; n++)
    {
      *zptr++ = (z = nextLine(z,&znr));
      *pnr++  = znr;
    }

    /* Now redisplay the window: */
    printAll();
  }
  else
  {
```

```
    /* Scrolling: */
    ScrollRaster(actualEditor->rp,
             0L,num * (LONG)actualEditor->ch,
             (LONG)actualEditor->xoff,(LONG)actualEditor->yoff,
             (LONG)actualEditor->xscr,(LONG)actualEditor->yscr);

    /* zlinesptr/nr recalculated: */
    zptr = actualEditor->zlinesptr;
    zold = &(ZLINESPTR(num));
    pnr = actualEditor->zlinesnr;
    oldnr = &(ZLINESNR(num));
    for (n = 0; n <= wch - num; n++)
    {
      *zptr++ = *zold++;
      *pnr++  = *oldnr++;
    }

    z = *--zold;
    zold = zptr - 1;       /* mark for output! */
    znr = *--oldnr;
    oldnr = pnr - 1;
    while (n <= wch)
    {
      *zptr++ = (z = nextLine(z,&znr));
      *pnr++ = znr;
      n++;
    }

    /* Output new line: */
    for (n = num, y = actualEditor->yoff+(wch-num)*actualEditor-
>ch;
         (n && (*zold != NULL)); n--, zold++, y += actualEditor-
>ch)
      printLine(*zold,y);
  }

  actualEditor->toppos += num;
}

/*****************************
 *
 * scrollDown(num):
 *
 * Scroll the window contents
 * num lines down.
 * Num lines must
 * exist!
 *
 * num = number of lines.
 *
 *****************************/

void scrollDown (num)
register UWORD num;
{
  register struct Zline *z,**zptr,**zold;
  register UWORD n,wch = actualEditor->wch,y;
  UWORD znr,*pnr,*oldnr;

  if (num >= wch)
  {
    /* Display window without Scrolling */
```

```
    /* search backwards fo rtop line: */
    for (z = ZLINESPTR(0), znr = ZLINESNR(0), n = num - wch; n;
n--)
      z = prevLine(z,&znr);

    /* z points to first line after bottom windo edge. */
    for (n = 0, zptr = &(ZLINESPTR(wch)), pnr =
&(ZLINESNR(wch));
        n <= wch; n++)
    {
      *zptr-- = z;
      *pnr-- = znr;
      z = prevLine(z,&znr);
    }

    /* Now redisplay the window: */
    printAll();
  }
  else
  {
    /* Scrolling: */
    ScrollRaster(actualEditor->rp,
             0L,-num * (LONG)actualEditor->ch,
             (LONG)actualEditor->xoff,(LONG)actualEditor->yoff,
             (LONG)actualEditor->xscr,(LONG)actualEditor->yscr);

    /* zlinesptr recalculated: */
    zptr = &(ZLINESPTR(wch));
    zold = &(ZLINESPTR(wch - num));
    pnr = &(ZLINESNR(wch));
    oldnr = &(ZLINESNR(wch - num));
    for (n = 0; n <= wch - num; n++)
    {
      *zptr-- = *zold--;
      *pnr-- = *oldnr--;
    }

    for (z = *++zold, znr = *++oldnr, n = num; n; n--)
    {
      *zptr-- = (z = prevLine(z,&znr));
      *pnr-- = znr;
    }

    /* Output new line: */
    for (n = num, y = actualEditor->yoff; (n && (*zold !=
NULL));
        n--, zold++, y += actualEditor->ch)
      printLine(*zold,y);
  }

  actualEditor->toppos -= num;
}

/*************************************
 *
 * cursorLeft:
 *
 * moce cursor one position
 * left, False returned if cursor,
 * is in first column.
 *
 *************************************/
```

```
BOOL cursorLeft()
{
  if (actualEditor->xpos > 1)
  {
    actualEditor->xpos--;
    if (actualEditor->xpos <= actualEditor->leftpos)
      scrollRight((UWORD)1);

    return (TRUE);
  }
  else
    return (FALSE);
}

/*****************************************
 *
 * cursorRight:
 *
 * moce cursor one position
 * right, False returned if cursor,
 * is in last column.
 *
 *****************************************/

BOOL cursorRight()
{
  if (actualEditor->xpos < MAXWIDTH)
  {
    actualEditor->xpos++;
    if (actualEditor->xpos
      >= (actualEditor->leftpos + actualEditor->wcw))
      scrollLeft((UWORD)1);

    return (TRUE);
  }
  else
    return (FALSE);
}

/*****************************************
 *
 * cursorUp:
 *
 * Move cursor one line up.
 * False returned when
 * on first line.
 *
 *****************************************/

BOOL cursorUp()
{
  UWORD help;

  if (actualEditor->wdy)
    actualEditor->wdy--;
  else
    if (prevLine(ZLINESPTR(0),&help))
      scrollDown((UWORD)1);
    else
      return (FALSE);
```

```
  actualEditor->ypos = ZLINESNR(actualEditor->wdy);

  return (TRUE);
}

/*****************************************
 *
 * cursorDown:
 *
 * Move cursor one line down.
 * False returned when
 * on last line.
 *
 *****************************************/

BOOL cursorDown()
{
  if (ZLINESPTR(actualEditor->wdy + 1))
  {
    if (++actualEditor->wdy >= actualEditor->wch)
    {
      scrollUp((UWORD)1);
      actualEditor->wdy--;
    }
    actualEditor->ypos = ZLINESNR(actualEditor->wdy);

    return (TRUE);
  }
  else
    return (FALSE);
}

/*********************************
 *
 * halfPageUp:
 *
 *  Move cursor on half page
 * up. The text is scrolled
 * half page up
 *
 * False returned when
 * in first line.
 *
 *********************************/

BOOL halfPageUp()
{
  register UWORD max,n;
  register struct Zline *z;
  UWORD help;

  max = actualEditor->wch / 2;
  for (n = 0, z = ZLINESPTR(0); n < max; n++)
    if (!(z = prevLine(z,&help)))
      break;

  if (n)
  {
    scrollDown(n);
    actualEditor->ypos = ZLINESNR(actualEditor->wdy);

    return (TRUE);
```

```
  }
  else
    return (FALSE);
}

/*********************************
 *
 * halfPageDown:
 *
 *  Move cursor on half page
 * down. The text is scrolled
 * half page up
 * oben gescrollt.
 * False returned when
 * on last line.
 *
 *********************************/

BOOL halfPageDown()
{
  register UWORD max,n;
  register struct Zline *z;
  UWORD help;

  max = actualEditor->wch / 2;
  for (n = 0, z = ZLINESPTR(actualEditor->wch - 1); n < max; n++)

    if (!(z = nextLine(z,&help)))
      break;

  if (n)
  {
    scrollUp(n);
    actualEditor->ypos = ZLINESNR(actualEditor->wdy);

    return (TRUE);
  }
  else
    return (FALSE);
}

/*******************************
 *
 * handleKeys(buf,len):
 *
 * Handles keyboard messages.
 *
 * buf ^ buffer with character.
 * len = number of characters.
 *
 *******************************/

void handleKeys(buf,len)
register UBYTE  *buf;
register WORD     len;
{
  register UBYTE first;
  register struct Zline *z;

  while (len > 0)
  {
    first = *buf;
```

```
    buf++;
    len--;
    if ((first == CSI) && (len > 0))
    {
      len--;
      switch (*buf++)
      {
        case CUU:
          cursorUp();
          break;
        case CUD:
          cursorDown();
          break;
        case CUF:
          cursorRight();
          break;
        case CUB:
          cursorLeft();
          break;
        case SU:
          halfPageDown();
          break;
        case SD:
          halfPageUp();
          break;

        default:
          buf--;
          len++;
          break;
      }
    }
    else if ((first == CFOLD) && (actualEditor->maxfold <
ZLF_FOLD))
    {
      actualEditor->maxfold++;
      if (z = ZLINESPTR(actualEditor->wdy))
        if ((z = z->succ)->succ)
          /* If there is a following line: */
          if (((ZLINESPTR(actualEditor->wdy)->flags & ZLF_FOLD)
              < (z->flags & ZLF_FOLD))
            && ((z->flags & ZLF_FOLD) == actualEditor->maxfold))
          {
            /* When this begins a new fold: */
            actualEditor->minfold = actualEditor->maxfold;
            ZLINESPTR(0)   = z;
            ZLINESNR(0)    = ZLINESNR(actualEditor->wdy) + 1;
            actualEditor->wdy = 0;
          }

      restoreZlinenptr();
      printAll();
      actualEditor->ypos = ZLINESNR(actualEditor->wdy);
    }
    else if ((first == CENDF) && (actualEditor->maxfold))
    {
      actualEditor->maxfold--;
      if (actualEditor->minfold > actualEditor->maxfold)
        actualEditor->minfold = actualEditor->maxfold;

      if (((ZLINESPTR(0)->flags & ZLF_FOLD) > actualEditor-
>maxfold)
```

```
      || ((ZLINESPTR(0)->flags & ZLF_FOLD) < actualEditor-
>minfold))
      {
        ZLINESPTR(0) = prevLine(ZLINESPTR(0),&(ZLINESNR(0)));
        actualEditor->wdy = 0;
      }

      restoreZlinenptr();
      printAll();
      actualEditor->ypos = ZLINESNR(actualEditor->wdy);
    }
    else
      putchar(first);
  }
}
```

Here are a few specifics about some of the functions:

- The `defines` determine the characters controlling cursor movement and folding. The labels are taken from AmigaDOS sources. You must be careful with the check in `handleKeys` that the cursor movement character sequences are two to three characters long. For now a CSI comes from the characters given in the `defines`.

- The `restoreLinesptr` function recalculates the `linesptr` and the `zlinesnr` arrays. This function is necessary to enable redisplay of window contents after a fold is encountered.

- The scroll functions allow scrolling by more than one line or character. They allow more characters or lines to be scrolled than can fit on the screen at a time. The vertical scroll functions work well for searching for a specific text.

- A `for` loop operates in the scrolling functions instead of using the `printAll` function in the scroll functions. `printAll` erases the borders that are unnecessary to scrolling, and takes too much time.

- The `Zlinesptr` and the `zlinesnr` arrays must be recalculated for `scrollUp` and `scrollDown`. To avoid recalculating the entire array, the program moves the contents needed after the scrolling, just as the lines are scrolled.

- In addition to testing the folding, `cursorUp` and `cursorDown` must check to see if the next or previous line exists in the current level. The `nextLine` and `prevLine` functions must be expanded correspondingly to perform this task. These functions supply a pointer to the current line number, which must then be raised or lowered appropriately.

- To test the vertical scrolling for more than one line, the `halfPageUp` and `halfPageDown` functions scroll the text a

half window page up or down. The cursor position relative to the window remains unchanged.

- When the user presses <Ctrl><f>, the editor must determine whether the cursor is at a fold start mark. The fold level of the current line is less than the level of the following line within the fold (assuming that a following line exists). The following line is not checked with `nextLine` because the next line appears in the text, regardless of the fold. When the level of the following line corresponds to the maximum fold level plus one, the editor handles the current line as a fold start mark because the next line is invisible.

- When a fold is encountered, it must be determined that the above line does not belong to the fold just exited before calling `restoreLinesptr`. Otherwise the window contents make no sense.

Now we can expand the other modules. Two new functions must be declared in the main `Editor` module:

```
void handleKeys(),Cursor();
```

The `nextLine` and `prevLine` functions must be modified for the folding:

```
struct Zline *nextLine(line,pnr)
register struct Zline *line;
register UWORD          *pnr;
{
  register struct Zline *z;
  register UWORD minfold = actualEditor->minfold;
  register UWORD maxfold = actualEditor->maxfold;

  if (z = line)
    if (z->succ)
      do
      {
        if ((z = z->succ)->succ)
          if ((z->flags & ZLF_FOLD) < minfold)
          {
            /* Following lines lie outside of fold */
            z = NULL;
            break;
          }
          else
            *pnr += 1;
        else
        {
          /* no following line */
          z = NULL;
          break;
        }
      }
      while ((z->flags & ZLF_FOLD) > maxfold);
    else
      z = NULL;
```

```
  return (z);
}
```

The function `prevLine` looks exactly like it, only "`->succ`" is replaced throughout by "`->pred`":

```
struct Zline *prevLine(line,pnr)
register struct Zline *line;
register UWORD          *pnr;
{
  register struct Zline *z;
  register UWORD minfold = actualEditor->minfold;
  register UWORD maxfold = actualEditor->maxfold;

  if (z = line)
    if (z->pred)
      do
      {
        if ((z = z->pred)->pred)
          if ((z->flags & ZLF_FOLD) < minfold)
          {
            /* previous line lies outside fold */
            z = NULL;
            break;
          }
          else
            *pnr += 1;
        else
        {
          /* no previous line */
          z = NULL;
          break;
        }
      }
      while ((z->flags & ZLF_FOLD) > maxfold);
    else
      z = NULL;

  return (z);
}
```

The `OpenEditor` function initializes the other elements of the editor:

```
ed->leftpos = 0;
ed->wdy     = 0;
ed->minfold = 0;
ed->maxfold = 0;

/* zlinesptr/nr-Array initialization: */
      for (n = 0, zptr = ed->zlinesptr, pnr = ed->zlinesnr;
          n < MAXHEIGHT; n++, zptr++, pnr++)
      {
        *zptr = NULL;
        *pnr  = 1;
      }
```

The main program now requires two new variables in place of `SHORT l`:

```
ULONG mouseX,mouseY
```

The main loop also requires some changes. Here is the current listing of the main loop:

```
do
  {
    /* Set cursor : */
    Cursor();

    signal = Wait(1L << edUserPort->mp_SigBit);

    /* erase cursor again: */
    Cursor();

    while (imsg = GetMsg(edUserPort))
    {
      class     = imsg->Class;
      code      = imsg->Code;
      qualifier = imsg->Qualifier;
      iaddress  = imsg->IAddress;
      mouseX    = imsg->MouseX;
      mouseY    = imsg->MouseY;

      ReplyMsg(imsg);

      /* Event processing: */
      switch (class)
      {
        case RAWKEY:
          if (! (code & IECODE_UP_PREFIX))
          {
            inputEvent.ie_Code                = code;
            inputEvent.ie_Qualifier = qualifier;
            if ((inputLen = RawKeyConvert(
                  &inputEvent,inputBuffer,MAXINPUTLEN,NULL)
                  ) >= 0)
              handleKeys(inputBuffer,inputLen);
          }
          break;

        case MOUSEBUTTONS:
          {
            register WORD x,y;

            if (mouseX <= actualEditor->xoff)
              x = 0;
            else
              x = (mouseX - actualEditor->xoff)
                / actualEditor->cw;
            if (++x >= actualEditor->wcw)
              x = actualEditor->wcw - 1;
            x += actualEditor->leftpos;
            if (x > MAXWIDTH)
              x = MAXWIDTH;

            if (mouseY <= actualEditor->yoff)
              y = 0;
            else
              y = (mouseY - actualEditor->yoff)
                / actualEditor->ch;
```

```
            if ((y < actualEditor->wch)
              && (actualEditor->zlinesptr[y]))
            {
              actualEditor->xpos = x;
              actualEditor->wdy  = y;
              actualEditor->ypos = actualEditor->zlinesnr[y];
            }
          }

        case NEWSIZE:
          initWindowSize(actualEditor);

          /* check if cursor still in window! */
          if (actualEditor->xpos > actualEditor->leftpos
                              +actualEditor->wcw)
            actualEditor->xpos = actualEditor->leftpos
                              + actualEditor->wcw;

          if (actualEditor->wdy >= actualEditor->wch)
            actualEditor->ypos = actualEditor->zlinesnr
              [(actualEditor->wdy  = actualEditor->wch - 1)];

          break;

        case REFRESHWINDOW:
          BeginRefresh(actualEditor->window);
          printAll();
          EndRefresh(actualEditor->window,TRUE);
          break;

        case CLOSEWINDOW:
          running = FALSE;
          break;

        default:
          printf("Not a processable event: %lx\n",class);

      } /* of case */
    } /* of while (GetMsg()) */
  } while (running);
```

The MOUSEBUTTONS check lets the user place the mouse on the window by clicking on that screen character, as with most word processors.

The `memory` module remains unchanged. However, the `output` module must be modified to display fold markers in alternate colors. First, a `printLine` needs a new global variable to make sure the last column is not displayed:

```
/*********************
 *
 * Global Variables:
 *
 *********************/

UWORD SplitDec = 0;
```

The `initWindowSize` function must recalculate the `zlinesptr` and the `zlinesnr` arrays. This requires the definition of two additional variables:

```
UWORD znr,*pnr·
```

The array calculation looks like this:

```
/* Now restore the zlineptr field: */
if (newh != ed->wch)
{
  if (newh < ed->wch)
  {
    /* Pointer to Null set: */
    zptr = ed->zlinesptr + newh + 1;
    for (n = newh; n < ed->wch; n++)
      *zptr++ = NULL;
  }
  else
  {
    zptr  = ed->zlinesptr + ed->wch;
    z     = *zptr++;
    pnr = &(ed->zlinesnr[ed->wch]);
    znr   = *pnr++;
    for (n = ed->wch; n < newh; n++)
    {
      *zptr++ = (z = nextLine(z,&znr));
      *pnr++  = znr;
    }
  }

  ed->wch = newh;
}
```

The `convertLineForPrint` function needs changes as well. Output must begin in the column determined by `leftpos`, not just the first column of the window. A counter (`skip`) to `leftpos` does this check, first seeing if `skip` has already reached zero. If `skip` equals zero the character is written. Otherwise, `skip` decrements by one. We skip the first `skip` character:

```
register UWORD skip = actualEditor->leftpos;
```

The determination of `lastchar` stays the same, but the conversion is changed:

```
/* line conversion: */
tab = actualEditor->tabstring;
while ((len--) && (l < w))
  if ((*buf = *line++) == TAB)
    do
    {
      tab++;
      if (skip)
        skip--;
      else
      {
        l++;
        *buf++ = ' ';
```

```
      }
    } while ((*tab) && (l < w));
  else
  {
    tab++;
    if (skip)
      skip--;
    else
    {
      l++;
      buf++;
    }
  }
```

The rest of the function is unchanged.

The `printAt` function must have additional parameters dealing with character color. These parameters are `FGPEN` or `FOLDPEN`. Both parameters are important since after a space, text color cannot be sure whether or not the text is a fold mark. The foreground color must be changed for spaces, and then reset:

```
void printAt   (buf,len,x,y,fgpen)
register UBYTE  *buf;
WORD                 len;
register UWORD        x,y;
ULONG                    fgpen;
```

Throughout this function

```
SetAPen(rp,FGPEN);
```

is replaced by:

```
SetAPen(rp,fgpen);
```

The `printLine` function can be expanded so that it recognizes fold marks and changes text color accordingly. `FGPEN` must be disabled at the end of the function so that we can leave the program at any place where the foreground color is `FGPEN`:

```
void printLine       (line,y)
register struct Zline *line;
register UWORD             y;
{
  static UBYTE buf[MAXWIDTH];
  register UWORD w;
  register struct Zline *z;
  register ULONG pen;

  convertLineForPrint(line+1,line->len,w = actualEditor-
>wcw,buf);

  /* If start/end marker of fold => FOLDPEN */
  if (line->flags & ZLF_FSE)
  {
    if ((z = line->succ)->succ)
    {
```

```
      if ((z->flags & ZLF_FOLD) <= actualEditor->maxfold)
      {
        SetAPen(actualEditor->rp,FOLDPEN);
        pen = FOLDPEN;
      }
      else
        pen = FGPEN;
    }
    else
    {
      SetAPen(actualEditor->rp,FOLDPEN);
      pen = FOLDPEN;
    }

    printAt(buf,w - SplitDec,actualEditor->xoff,y,pen);

    /* Old write color first again: */
    if (pen != FGPEN)
      SetAPen(actualEditor->rp,FGPEN);
  }
  else
    printAt(buf,w - SplitDec,actualEditor->xoff,y,FGPEN);
}
```

Changes must still be made to the `test.c` module before we can test the new extensions. The line number displayed by the `print` function comes from the `zlinesnr` array:

```
void print()
{
  register struct Zline **z;
  register UWORD n,*pnr;

  for (n = 0, z = actualEditor->zlinesptr,
      pnr = actualEditor->zlinesnr;
      n <= actualEditor->wch; n++, z++, pnr++)
    printf("Zline %d an Adresse %6lx.\n",*pnr,*z);
}
```

The number of lines should be adapted for the input and delete line functions, by adding the following for `input`:

```
actualEditor->num_lines++;
```

and for `delete_line`:

```
actualEditor->num_lines--;
```

The `init_zlinesptr` function must determine the fold level of all of the lines:

```
void init_zlinesptr()
{
  register struct Zline *z,**zptr;
  register UWORD n,*pnr,fold = 0;
  UWORD znr;

  for (z = actualEditor->zlines.head; z->succ; z = z->succ)
  {
```

```
    z->flags = ((z->flags & ~ZLF_FOLD) | (fold & ZLF_FOLD));
    if (z->len >= 8)
    {
      if ((*((ULONG *)(z+1))=='/*#F') && (*((ULONG
*)(z+1)+1)=='OLD:'))
      {
        fold++;
        z->flags |= ZLF_FSE;
      }
      if ((*((ULONG *)(z+1))=='/*#E') && (*((ULONG
*)(z+1)+1)=='NDFD'))
      {
        fold--;
        z->flags |= ZLF_FSE;
      }
    }
  }

  for (n = 0, z = actualEditor->zlines.head,
         zptr = actualEditor->zlinesptr,
         pnr = actualEditor->zlinesnr, znr = 1;
         n < MAXHEIGHT; n++)
  {
    *zptr++ = z;
    *pnr++  = znr;
    z = nextLine(z,&znr);
  }
}
```

The fold mark test looks rather "wild," but it works faster than the string compare function from the standard library. This happens because four characters are constantly compared with one another (ULONG = 4 bytes). Currently it works in our test module because we can allow ourselves some leeway in test programming. Later, if we allow any other strings as fold marks, we use the `strcmp` or `strncmp` functions, because there is no guarantee that the corresponding character string is exactly eight characters in length.

A `printAll` call at the end of the function test ensures that the window contents are updated. Compile and link this version of the editor.to test this version of the editor. We will use the Amiga's ability to redirect input from a file. The optionl diskette contains a `TestText` file to do this. The file contains a linefeed, an a, and another linefeed at the end of each line. Using redirection, this file can be loaded into our test version of the editor using the `append` (a) command. An escape character and a linefeed are located at the end of the file to let the test editor know when we are finished appending text. Please see Apendix A for complete information on altering a standard text file suitable for redirection into our test editor.

You'll find the `TestText` source, the editor source text and compiled editor in the `V0.4` directory of the optional disk.

After you have created a test file, enter the following to start the editor and use the `TestText` source as redirected input, see the optional diskette or Appendix A for the `TestText` listing:

```
Editor <TestText
```

The end of `TestText` contains an escape code, which indicates the end of the source file. You can now examine the editor and some of its features, including folding.

Debugging

The debugger is an important part of C development. The only way to become proficient at using the debugger is to sit down and use it. Therefore, we have an exercise to help you learn debugging: The cursor's Y position should be displayed from the debugger.

Prepare to recompile the editor, but this time enter the following:

```
make debug
```

The `makefile` ensures that the finished program contains information about variable names and symbol tables. This helps the programmer find certain locations in the program. After compilation, enter the following to start the debugger from the Aztec disks, this depends on how you have installed the Aztec C compiler, db may be in a different directory:

```
SYS3:bin/db
```

This loads a file named `.dbinit` that contains only one instruction (`al`). This lets you begin a new task while in the debugger. Start the editor as explained above. The debugger notices that a new task is in operation. It checks to see if the program contains a symbol table, and loads the symbol table if it exists.

After all loading ends, the debugger displays the first instruction:

```
jmp .begin
```

We want the debugger to display the Y position of the cursor. This display occurs only after cursor movement. The Y position cursor movement occurs primarily in the `cursorUp` and `cursorDown` functions. We must know the location in memory containing the position number, and access this location for the Y position display. The Y position is an element of the `Editor` structure, and the `cursorUp` and `cursorDown` functions change this position (see the disassembled list below for functions in detail). Enter the following debugger instruction:

```
u cursorUp
```

The debugger displays the memory in disassembled form from the address `cursorUp` on. The list looks something like this:

```
_cursorUp          link    a5,#-2
_cursorUp+4        movea.l _actualEditor,a0
_cursorUp+8        tst.w   252(a0)
_cursorUp+c        beq.s   _cursorUp+18
_cursorUp+e        movea.l _actualEditor,a0
```

```
_cursorUp+12        subq.w  #1,252(a0)
_cursorUp+16        bra.s   _cursorUp+40
_cursorUp+18        pea     -2(a5)
_cursorUp+1c        movea.l _actualEditor,a0
_cursorUp+20        move.l  ca(a0),-(a7)
_cursorUp+24        jsr     _prevLine
_cursorUp+28        addq.w  #8,a7
_cursorUp+2a        tst.l   d0
_cursorUp+2c        beq.s   _cursorUp+3a
_cursorUp+2e        move.w  #1,-(a7)
_cursorUp+32        jsr     _scrollDown
_cursorUp+36        addq.w  #2,a7
_cursorUp+38        bra.s   _cursorUp+40
_cursorUp+3a        moveq   #0,d0
_cursorUp+3c        unlk    a5
_cursorUp+3e        rts
_cursorUp+40        movea.l _actualEditor,a0
_cursorUp+44        moveq   #0,d0
_cursorUp+46        move.w  252(a0),d0
_cursorUp+4a        asl.l   #1,d0
_cursorUp+4c        movea.l d0,a1
_cursorUp+4e        adda.l  _actualEditor,a1
_cursorUp+52        movea.l _actualEditor,a6
_cursorUp+56        move.w  lca(a1),250(a6)
_cursorUp+5c        moveq   #1,d0
_cursorUp+5e        bra.s   _cursorUp+3c
```

The `zlinesnr` array supplies the Y position. We're looking for a program section that reads a value from an array, then writes the value as an element of the `Editor` structure. This section appears at the address `_cursorUp+40`. First a pointer to the current `Editor` structure is needed after A0. Then `wdy` passes to D0; this value becomes the index of the `zlinesnr` array (thus the logical shift left multiplication (`asl`)). The value of the array passes to the `ypos` variable at address `_cursorUp+56`. That means that `ypos` lies 250 hex bytes from the beginning of the structure.

Some knowledge in assembler is required to discover which assembler memory location represents which C variable. Another way of obtaining this value: Count up the elements in the `Editor` structure, according to their memory needs in bytes. Once we determine the location of `ypos`, we must allocate an area at which we can set a breakpoint, which ensures display of the Y position. After the Y position changes, the program jumps to address `_cursorUp+3c`. This address contains the UNLK assembler instruction, which concludes every assembler subroutine in Aztec C. Set the breakpoint at the following address:

```
bs cursorUp+3c ;pd 250+*actualEditor;g
```

About the arguments in the above line: The `pd` (Print Decimal) forces the debugger to display the contents of the given memory location in decimal notation). The memory location corresponds to the Y position of the current editor, 250 bytes after the address to which the current editor points. The `g` means that the debugger should restart the editor. We don't want to stop the editor, but instead want to have the Y

position displayed. Set the next breakpoint in the `cursorDown` subroutine, and at the end of the program. Enter:

```
u cursorDown
```

Continue to enter `u` until you find the `UNLK` instruction. It will appear at address `_cursorDown+66`. Set the breakpoint as before:

```
bs cursorDown+66 ;pd 250+*actualEditor;g
```

This concludes the preparations. Press <g> (GO) to start the editor. Everything runs as usual, until you press the <Cursor up> key. Then the debugger window comes to the foreground and activates. The cursor does not appear in the debugger window following a question mark. One catch: The editor is no longer in the foreground, nor is it active.

Reduce the size of the debugger window so that it doesn't block the editor window but is still large enough to clearly display the Y position. Enlarge the editor window so that you can see the entire debugger window. Now reactivate the editor window by either clicking on the window border or the cursor (clicking anywhere else moves the cursor to another position).

Every time you press the <Cursor up> or <Cursor down> key, you must reactivate the editor window (this is tolerable, but not very elegant). Move the cursor up a couple lines onto folds, and see if the position displayed is true. Enter the fold and scroll around a little. Once you have done some scrolling, return to the beginning of the text and see if this has the line number one. You'll find an error in the calculation of the Y position.

With some logical thought we can determine how the error occurred. Everything runs properly, as long as we remain in the window without accessing any folds. This is fine, since line numbers are stored in the `zlinesnr` array. As soon as this array changes, however, an error occurs. The `nextLine` function (which calculates the line numbers) initializes the line numbers, returning zero. Two options are available for seeking errors:

1.) List and read the source text until you find the error;

2.) Continue executing the program through the debugger and narrow the error range down until you can easily locate the error.

Let's start by searching the source. Load the `cursor.c` module into an editor and look at the program locations which affects folding. You won't see any obvious errors there—we didn't. Since the problem seems to occur when scrolling, search for the `scrollUp` and `scrollDown` functions. They don't seem to contain any errors, either. Before we continue testing, look at the `prevLine` function, which is accessed by the `scrollDown` function. Notice that in this function the line

numbers increment by one (exactly like in `nextLine (*pnr++)`), instead of decrement by one (`*pnr--`).

Change this, save the file, and recompile the editor. The editor in directory V0.4 on the optional diskette contains this error, so you can check it with the debugger. When the error is removed, you can test the editor again if you have the desire.

4.3.8 Text Input

The editor should be able to handle text entry and editing. Functions listed in this section will give the editor capabilities of entering, inserting and deleting characters and lines in a source text. Let's begin with the "how" of writing these functions.

Insert mode

Before inserting a character, the corresponding line must be copied into the buffer of the `Editor` structure. Then all of the characters under the cursor or to the right of the cursor are moved to the right, and the new character appears at the current cursor position. This occurs in insert mode only. If the user tries to insert characters past the right border of the screen, the screen flashes: You cannot move past the right border of the full size window.

Overwrite mode

When the editor is in overwrite mode, any characters entered from the keyboard are placed in the text. If any characters already exist in the same line, the new characters replace the old. The cursor moves one character to the right for each new character.

Control characters play a special role when inserting, particularly CR (carriage return = 13), LF (linefeed = 10), BS (Backspace = 8), DEL (Delete = 127), and TAB (9). CR and LF create a new line at the current cursor position. BS and DEL delete characters. TAB inserts multiple spaces.

Tabs

Tabs require special attention. Since the editor's main purpose is for entering program text, it doesn't really matter whether tabs are spaces or not. The Aztec C compiler treats tabs as blank spaces. Consider the following:

- When we copy a line to the buffer, the tabs change into spaces, just as we did in the `convertLineForPrint` function.
- When the line transfers from the buffer back into the program line, the spaces change back into tabs.

The two items mentioned above offer formatting with a minimum of memory requirements. The sole disadvantage of this method: No

multiple spaces (the editor views these as true spaces, not tabs) or user-specified tabs allowed. A flag named `actualEditor->tabs` solves this problem. When this flag is set to one, spaces convert to tabs. When it is set to zero, tabs are treated as normal control characters (<Ctrl><i>) for input and output (i.e., no conversion).

Now we must write the buffer back into the line structure. This occurs when the cursor moves out of the current line. The function needed executes when a message is processed within the editor's main loop. This function writes the buffer contents back to the line if the cursor exits this line. A variable named `bufypos` either contains the number of the line found in the buffer, or contains zero if no lines exist in the buffer. The variable `actual` points to the line structure of the line found in the buffer. In addition, the `ZLF_USED` flag in the line structure must be set when copying a line in the buffer. This flag determines whether a line from the output appears in the buffer.

Now we can begin to create the specifics of the new `Edit.c` module:

```
void getLineForEdit   (line,znr)
register struct Zline *line;
register UWORD            znr;
```

This function copies the line `line` to the buffer and sets all of the corresponding variables. The `convertLineForPrint` function performs line conversion. The `actualEditor->leftpos` must have been preset to zero, allowing the conversion of all characters and the return to original values later. This function requires some revision—the converted line's length is needed for placing in `actualEditor->buflen`, and we must know `lastchar` (from `actualEditor->buflastchar`). For character deletion, the last character in the buffer must be replaced by `lastchar`, so that the line displays correctly on the screen.

```
BOOL saveLine(line)
struct Zline *line;
```

Save the line found in the buffer to the line structure, which points to `actualEditor->actual`. Eventually a new line must be used in case the length of the original line differs from the new line. In case the old or the changed line represents a fold marker, the levels of all of the following lines must correspond to the old or changed line. If the line cannot be rewritten, FALSE returns.

```
void saveIfCursorMoved()
```

The main loop of the editor calls this function after processing each message. This function determines whether the cursor has left the actual input line, and. if so, calls the `saveLine` function.

The following functions control text processing. Since these are implemented as individual functions, the implementation of a command language can be added fairly easily, because individual instructions only

call their corresponding functions. Each function returns FALSE if it cannot be executed.

```
BOOL undoLine()
```

When a line is found in the buffer, this removes changes and displays the original line.

```
BOOL deleteChar()
```

Erases the character under the cursor and moves the rest of the line one character to the left (enabled by pressing the <Delete> key).

```
BOOL backspaceChar()
```

Erases the character to the left of the cursor and moves the rest of the line one character to the left provided insert mode is enabled. If insert mode is disabled, a space appears to the left of the cursor, and the cursor moves one character to the left.

```
BOOL insertChar(c)
UBYTE c;
```

Inserts character `c` at the current cursor position provided insert mode is enabled. If insert mode is disabled, the new character appears at the current cursor position. This function also processes tabs.

```
BOOL insertLine(c)
UBYTE c;
```

Inserts a linefeed at the current cursor position. If the cursor is in the middle of a line, this line breaks into two lines at the cursor. Character `c` is either CR or LF, according to which key was pressed. The cursor moves to the beginning of the next line.

The `insertLine` function acted as a basis for auto indent. Auto indent moves the cursor to the starting column of the new line after pressing the <Return> key. That is, if you started a line in column six of a line and pressed <Return> while auto indent is active, the cursor moves to column six of the new line. This also occurs if you press <Return> to break a line while the cursor is in the middle of the line. Resetting the `autoindent` flag of the `Editor` structure disables auto indent.

```
BOOL deleteLine()
```

Erases the line in which the cursor currently lies. You must be careful if the cursor lines in a fold at the end of a text. If you try to delete all of the lines, other folded lines still exist. In this case this function reduces `minfold` until the line found appears in the top line of the window. The function tries to leave the cursor at the current position, scrolling up the bottom section of the window's contents after the line is deleted.

All of the functions are called from the `handleKeys` function, which appeared in Section 4.3.7. The source text of our new module looks like this:

```
<src/Edit.c;>

/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"

/***********
 *
 * Defines:
 *
 ***********/

#define CR 13
#define LF 10
#define TAB 9

/* Lenght of fold start to end */
#define FSE_LEN 10
/* Number of characters, significant for the fold markers: */
#define FSE_SIG 8

/**********************
 *
 * External Functions:
 *
 **********************/

UWORD convertLineForPrint(),strncmp(),recalcTopOfWindow();
void deleteZline(),Insert(),printAll(), restoreZlinenptr();
void DisplayBeep(),printLine(),AddHead(),scrollRight(),
scrollUp();
void ScrollRaster();
struct Zline *newZline(),*nextLine(),*prevLine();
BOOL cursorLeft(),cursorRight(),cursorDown(), cursorHome();

/*********************
 *
 * External Variables:
 *
 *********************/

extern struct Editor *actualEditor;

/*********************
 *
 * Global variables:
 *
 *********************/

UBYTE foldStart[] = "/*#FOLD:*/";
UBYTE foldEnde[]    = "/*#ENDFD*/";
```

```
/***************
 *
 * Functions:
 *
 ***************/

/*******************************
 *
 * getLineForEdit(line,znr)
 *
 * Copy line from editor to the
 * the buffer.
 *
 * line ^ Zline.
 * znr = line number of the line.
 *
 *******************************/

void getLineForEdit   (line,znr)
register struct Zline *line;
register UWORD            znr;
{
  register UWORD leftpos;

  /* get leftpos: */
  leftpos = actualEditor->leftpos;
  actualEditor->leftpos = 0;

  /* convert line: */
  actualEditor->buflen =
    convertLineForPrint(line+1,line->len,MAXWIDTH + 1,
      actualEditor->buffer,&actualEditor->buflastchar);

  /* Zline as "used"  : */
  line->flags |= ZLF_USED;
  actualEditor->actual = line;
  actualEditor->bufypos = znr;

  /* establish Leftpos: */
  actualEditor->leftpos = leftpos;
}

/*******************************
 *
 * cursorOnText:
 *
 * make sure that the cursor
 * is found on text.
 *
 *******************************/

void cursorOnText()
{
  register struct Zline **zptr;
  register UWORD l;

  /* makes sure that the cursor is really on text: */
  zptr = &(ZLINESPTR(l = actualEditor->wdy));
  while ((*zptr == NULL) && l)
  {
    zptr--;
```

```
    l--;
  }
  actualEditor->wdy  = l;
  actualEditor->ypos = ZLINESNR(l);
}

/************************
 *
 * deconvertLine():
 *
 * Deconver line.
 *
 * buf   ^ target buffer.
 * buffer^ source pointer.
 * buflen= the lenght.
 *
 ************************/

UWORD deconvertLine(buf, buffer,buflen)
UBYTE           *buf,*buffer;
UWORD                    buflen;
{
  register UBYTE *p1,*p2,*tab,*fb = NULL;
  register WORD l;

  /* deconvert line: */
  p1 = buffer;
  p2 = buf;
  tab= actualEditor->tabstring;
  l   = buflen;

  if (actualEditor->tabs)
    /* Spaces converted to tabs: */
    while (l)
    {
      if ((*p2 = *p1++) == ' ')
      {
        if (fb == NULL)
          fb = p2;
      }
      else
      {
        if (fb)
          fb = NULL;
      }

      /* fb points to first space              */
      /* other characters are copied. (first blank)    */
      if ((! *++tab) && (fb))
      {
        *fb = TAB;
        fb++;
        p2 = fb;
      }
      else
        p2++;

      l--;
    }
  else
    while (l)
    {
```

```
      *p2 = *p1;
      p1++;
      p2++;
      l--;
    }

  /* Spaces at end deleted: */
  if (actualEditor->skipblanks)
    while ((p2 > buf) && ((*(p2-1) == ' ') || (*(p2-1) == TAB)))
      p2--;

  return ((UWORD)(p2 - buf));
}

/****************************************
 *
 * getFoldInc(line)
 *
 * determine if line is fole start
 * or end mark and gives 1 or -1
 * back.
 * Else a 0 is returned.
 *
 * line ^ result line.
 *
 ****************************************/

UWORD getFoldInc(line)
struct Zline    *line;
{
  register UBYTE *p1;
  register UWORD l;

  /* fold marker at start of line! */
  p1 = (UBYTE *)(line + 1);
  l = line->len;
  while (l && ((*p1 == ' ') || (*p1 == TAB)))
  {
    p1++;
    l--;
  }

  /* Type of fold markers (-> l): */
  /*  1 = Start,              */
  /*  -1   End and              */
  /*  0 no fold marker.       */
  if (l >= FSE_SIG)
    if (strncmp(p1,foldStart,FSE_SIG) == 0)
      l = 1;
    else if (strncmp(p1,foldEnde,FSE_SIG) == 0)
      l = -1;
    else
      l = 0;
  else
    l = 0;  /* has no fold markers */

  return (l);
}

/*******************************
 *
 * saveLine(line)
```

```
 *
 * Save line.
 *
 * line ^ old line-Structure.
 *
 ******************************/

BOOL saveLine(line)
struct Zline *line;
{
  static UBYTE buf[MAXWIDTH + 2];              /* +2 for CR and
LF */
  register UBYTE *p1,*p2,*tab,*fb = NULL;
  register WORD l,len;
  struct Zline *pred,*old,**zptr;

  /* Only for security: */
  if (actualEditor->actual == NULL)
    return (FALSE);

  /* deconvert line: */
  p2 = buf + (len = deconvertLine(buf,actualEditor->buffer,
                          actualEditor->buflen));

  /* firm place, or line ends with  CR or LF, */
  /*  and increases len correspondingly. */
  p2 = buf + len;
  if (l = line->len)
  {
    p1 = ((UBYTE *)(line + 1)) + line->len - 1;
    if (*p1 == CR)
    {
      len++;
      *p2 = CR;
    }
    else if (*p1 == LF)
    {
      len++;
      if ((l > 1) && (*--p1 == CR))
      {
        len++;
        *p2++ = CR;
      }
      *p2 = LF;
    }
  }

  /* calculate new line: */
  if (EVENLEN(len) != EVENLEN(line->len))
  {
    old = line;
    if (line = newZline(len))
    {
      Insert(&actualEditor->zlines,line,old);
      line->flags = old->flags & ~ZLF_USED;
      deleteZline(old);

      /* now possible to convert pointer: */
      for (l = 0, zptr = actualEditor->zlinesptr;
          l <= actualEditor->wch; l++, zptr++)
        if (*zptr == old)
        {
```

```
            *zptr = line;
            break;
          }
      }
      else
        return (FALSE);
    }
    else
    {
      line->len    = len;
      line->flags &= ~ZLF_USED;
    }

    /* write line : */
    p1 = (UBYTE *)(line + 1);
    p2 = buf;
    l   = len;

    while (l)
    {
      *p1 = *p2;
      p1++;
      p2++;
      l--;
    }

    /* reconstruct folding: */
    /* Was old or is new line on fold mark? */
    l = getFoldInc(line);
    if ((line->flags & ZLF_FSE) || l)
    {
      /* next the ZLF_FSE  flag set to 1: */
      if (l)
        line->flags |= ZLF_FSE;
      else
        line->flags &= ~ZLF_FSE;

      /* calculate how the fold level of the following lines */
      /* : level += l.          */
      if ((old = line->succ)->succ)
      {
        if (((line->flags+1) & ZLF_FOLD) == (old->flags &
ZLF_FOLD))
          /* Old line was fld start: */
          l -= 1;
        else if ((line->flags & ZLF_FOLD) == ((old->flags+1) &
ZLF_FOLD))
          /* Old line was fold end: */
          l += 1;

        if (l)
        {
          /* calculate fold level fo rall following lines: */
          while (old->succ)
          {
            old->flags = (old->flags & ~ZLF_FOLD)
                       | ((old->flags + l) & ZLF_FOLD);
            old = old->succ;
          }

          restoreZlinenptr();
          printAll();
```

```
        cursorOnText();
      }
    }
  }

  /* line output: */
  /* First calculate Ypos:              */
  for (l = 0, zptr = actualEditor->zlinesptr;
      l <= actualEditor->wch; l++, zptr++)
    if (*zptr == line)
    {
      printLine(line,actualEditor->yoff + actualEditor->ch*l);
      break;
    }

  /* Delete reference: */
  actualEditor->actual = NULL;
  actualEditor->buflen   = 0;
  actualEditor->bufypos = 0;

  /* Text was changed! */
  actualEditor->changed = 1;

  return (TRUE);
}

/*************************************
 *
 * saveIfCursorMoved:
 *
 * Save line in buffer,
 * if the cursor is moved.
 *
 *************************************/

void saveIfCursorMoved()
{
  register UWORD bufypos;

  if (bufypos = actualEditor->bufypos)
    if (actualEditor->ypos != bufypos)
      if (! saveLine(actualEditor->actual))
        DisplayBeep(NULL);
}

/*****************************************
 *
 * undoLine:
 *
 *
 *
 *
 *****************************************/

void undoLine()
{
  register struct Zline *z;

  if (z = actualEditor->actual)
  {
    actualEditor->actual = NULL;
    actualEditor->bufypos = 0;
```

```
    actualEditor->buflen  = 0;

    z->flags &= ~ZLF_USED;

    /* line output: */
    printLine(z,actualEditor->yoff
            + actualEditor->ch*actualEditor->wdy);
  }
}

/************************************************
 *
 * getBufferPointer(pptr,pinc,prest):
 *
 * Initialize pointer to actualEditor->buffer.
 * The buffer is initialized.
 * FALSE is returned if the buffer
 * cannot be initialized.
 *
 * pptr ^ Pointer in buffer at cursor position.
 * pinc ^ Offset to buffer start.
 * prest^ buflen - inc.
 *
 ************************************************/

BOOL getBufferPointer(pptr,pinc, prest)
UBYTE              **pptr;
UWORD                *pinc,*prest;
{
  register UBYTE *ptr;
  register WORD inc,rest;

  /* Buffer initialized: */
  if (actualEditor->actual == NULL)
  {
    register UWORD wdy;

    if (ZLINESPTR(wdy = actualEditor->wdy))
      getLineForEdit(ZLINESPTR(wdy),ZLINESNR(wdy));
    else
      if (actualEditor->num_lines == 0)
      {
        /* first line: */
        register struct Zline *z;

        if (z = newZline((UWORD) 1))
        {
          *((UBYTE *)(z + 1)) = LF;
          AddHead(&actualEditor->zlines,z);
          ZLINESPTR(0) = z;
          ZLINESNR(0)  = (actualEditor->num_lines = 1);
          getLineForEdit(ZLINESPTR(0),ZLINESNR(0));
        }
        else
          return (FALSE);
      }
      else
        /* Cursor nott in line! */
        return (FALSE);
  }

  /* Calculate pointer to cursor position: */
```

```
  inc = actualEditor->xpos - 1;
  ptr = actualEditor->buffer + inc;
  rest= actualEditor->buflen - inc;

  /* What if inc > buflen? */
  if (inc < MAXWIDTH)
  {
    if (rest < 0)
    {
      register UBYTE *p;

      p = actualEditor->buffer + actualEditor->buflen;
      actualEditor->buflen -= rest;
      while (rest)
      {
        *p = ' ';
        p++;
        rest++;
      }
    }
  }
  else
    return (FALSE);

  *pptr = ptr;
  *pinc = inc;
  *prest= rest;
  return (TRUE);
}

/************************************
 *
 * deleteChar:
 *
 * Delete character under the cursor.
 * False is returned if no more
 * characters.
 *
 ************************************/

BOOL deleteChar()
{
  UBYTE *ptr;
  UWORD inc,rest;
  register UBYTE *p1,*p2;
  register UWORD l;

  if (! getBufferPointer(&ptr,&inc,&rest))
    return (FALSE);

  if (rest)
  {
    p1 = ptr + 1;
    p2 = ptr;
    l  = --rest;
    while (l)
    {
      *p2 = *p1;
      p1++;
      p2++;
      l--;
    }
```

```
    *p2 = actualEditor->buflastchar;

    /* Lenght - 1 */
    actualEditor->buflen--;

    printLine(actualEditor->actual,actualEditor->yoff
              + actualEditor->ch*actualEditor->wdy);

    return (TRUE);
  }
  else
    return (FALSE);
}

/*****************************
 *
 * backspaceChar:
 *
 * Delete character before cursor.
 * False is returned when
 * Cursor in start of line
 *
 *****************************/

BOOL backspaceChar()
{
  UBYTE *ptr;
  UWORD inc,rest;
  register UBYTE *p1,*p2;
  register UWORD l;

  if (! getBufferPointer(&ptr,&inc,&rest))
    return (FALSE);

  if (inc)
  {
    if (actualEditor->insert)
    {
      p1 = ptr;
      p2 = --ptr;
      l  = rest;
      while (l)
      {
        *p2 = *p1;
        p1++;
        p2++;
        l--;
      }
      *p2 = actualEditor->buflastchar;

      /* Length - 1 */
      actualEditor->buflen--;
      inc--;
    }
    else
    {
      *--ptr = ' ';
      inc--;
      rest++;
    }

    cursorLeft();
```

```
    printLine(actualEditor->actual,actualEditor->yoff
              + actualEditor->ch*actualEditor->wdy);

    return (TRUE);
  }
  else
    return (FALSE);
}

/****************************************
 *
 * insertChar(c)
 *
 * Inserts character in buffer.
 * FALSE returned if character
 * cannot be inserted.
 *
 * c= character.
 *
 ****************************************/

BOOL insertChar(c)
register UBYTE  c;
{
  UBYTE *ptr;
  WORD inc,rest;
  WORD xadd = 1;

  if (! getBufferPointer(&ptr,&inc,&rest))
    return (FALSE);

  if ((c == TAB) && (actualEditor->tabs))
  {
    if (actualEditor->insert)
    {
      /* Insert spaces: */
      register UBYTE *p1,*p2;
      register UWORD num,l;

      p1  = actualEditor->tabstring + actualEditor->xpos - 1;
      l  = MAXWIDTH - actualEditor->xpos;
      num = 1;
      while (l && (*++p1))
      {
        xadd++;
        l--;
        num++;
      }

      /* num = Number of spaces to insert: */
      if (actualEditor->buflen + num > MAXWIDTH)
        num = MAXWIDTH - actualEditor->buflen;

      /* Write rest of line : */
      p1 = ptr + rest;
      p2 = p1 + num;
      l   = rest;
      while (l)
      {
        *--p2 = *--p1;
        l--;
      }
```

```
            /* Insert spaces: */
            p1 = ptr;
            l  = num;
            while (l)
            {
              *p1 = ' ';
              p1++;
              l--;
            }

            actualEditor->buflen += num;
            rest += num;
          }
          else
          {
            /* move cursor only: */
            register UBYTE *tab;
            register UWORD max;

            tab = actualEditor->tabstring + actualEditor->xpos - 1;
            max = MAXWIDTH - actualEditor->xpos;
            while (max && (*++tab))
            {
              xadd++;
              max--;
            }
          }
        }
        else
        {
          /* besides a normal character: */
          if (actualEditor->insert)
            if (actualEditor->buflen < MAXWIDTH)
            {
              register UBYTE *p1,*p2;
              register UWORD l;

              p1 = ptr + rest;
              p2 = p1;
              l  = rest;
              while (l)
              {
                *p2 = *--p1;
                p2--;
                l--;
              }
              *ptr = c;

              /* Length + 1 */
              actualEditor->buflen++;
              rest++;
            }
            else
              /* Word-Wrap here later */
              return (FALSE);
          else
          {
            *ptr = c;
            if (rest = 0)
            {
              actualEditor->buflen++;
```

```
        rest = 1;
      }
    }
  }

  /* move cursor: */
  while (xadd)
  {
    if (! cursorRight()) break;
    xadd--;
  }

  /* output line: */
  printLine(actualEditor->actual,
    actualEditor->yoff + actualEditor->ch*actualEditor->wdy);

  return (TRUE);
}

/*************************
 *
 * insertLine(c):
 *
 * new line inserted before
 * actual line.
 * FALSE returned if
 * error encountered.
 *
 * c =  line end (CR/LF).
 *
 *************************/

BOOL insertLine(c)
UBYTE        c;
{
  static UBYTE buf[MAXWIDTH + 1];
  UBYTE *ptr;
  register UBYTE *p1,*p2;
  WORD inc,rest,len;
  UWORD autoindent;
  register UWORD l;
  register struct Zline *z;

  /* No new line inserted, when actuale line
    has no fold markers!!! */
  if ((z = actualEditor->actual) == NULL)
    z = ZLINESPTR(actualEditor->wdy);

  if (!z || !(z->flags & ZLF_FSE))
  {
    /* get line in buffer: */
    if (! getBufferPointer(&ptr,&inc,&rest))
      return (FALSE);

    /* calculate auto indent: */
    if (actualEditor->autoindent)
    {
      p1 = actualEditor->buffer;
      l = inc;
      while (l && ((*p1 == ' ') || (*p1 == TAB)))
      {
        p1++;
```

```
            l--;
          }

          if (l)
            autoindent = p1 - actualEditor->buffer + 1;
          else
            /* insert spaces in line: */
            autoindent = 1;
        }
        else
          autoindent = 1;

        /* deconvert line: */
        p2 = buf + (len = deconvertLine(buf,actualEditor-
>buffer,inc));
        *p2= c;
        len++;

        if (z = newZline(len))
        {
          z->flags = (actualEditor->actual->flags & ~ZLF_USED);

          p1 = (UBYTE *)(z + 1);
          p2 = buf;
          l = len;

          while (l)
          {
            *p1 = *p2;
            p1++;
            p2++;
            l--;
          }

          /* line connect: */
          Insert(&actualEditor->zlines,z,actualEditor->actual-
>pred);
          actualEditor->num_lines++;
          actualEditor->changed = 1;

          /* write rest of buffer line */
          /* remember auto indent! */
          p1 = actualEditor->buffer;
          l  = autoindent;
          while (--l)
            p1++;       /* Old contents remain! */

          p2 = ptr;
          l = rest;
          while (l)
          {
            *p1 = *p2;
            p1++;
            p2++;
            l--;
          }
          actualEditor->buflen = p1 - actualEditor->buffer;

          /* rest filled with lastchar: */
          l = inc + 1 - autoindent;
          while (l)
          {
```

```
  *pl = actualEditor->buflastchar;
  pl++;
  l--;
}

/* firm place, or new line fold marker: */
if (l = getFoldInc(z))
{
  register struct Zline *old;
  register UWORD fold;

  z->flags |= ZLF_FSE;

  /* calculate fold level for all following lines: */
  if ((old = z->succ)->succ)
    while (old->succ)
    {
      old->flags = (old->flags & ~ZLF_FOLD)
              | ((old->flags + l) & ZLF_FOLD);
      old = old->succ;
    }

  /* fit minfold and maxfold: */
  if ((l > 0) || ((z->flags & ZLF_FOLD) > 0))
  {
    fold = (z->flags & ZLF_FOLD) + l;
    if (actualEditor->minfold > fold)
      actualEditor->minfold = fold;
    if (actualEditor->maxfold < fold)
      actualEditor->maxfold = fold;

    ZLINESPTR(actualEditor->wdy) = z;
    actualEditor->wdy =
          recalcTopOfWindow(actualEditor->wdy);
  }
}

/* if Cursor calculate line => zlinesptr[0]: */
if (actualEditor->wdy == 0)
  ZLINESPTR(0) = z;

restoreZlinenptr();

/* redisplay window: */
actualEditor->xpos = autoindent;
if (actualEditor->xpos <= actualEditor->leftpos)
{
  /* If scrolling necessary: printAll */
  actualEditor->leftpos = actualEditor->xpos - 1;
  if (++actualEditor->wdy >= actualEditor->wch)
  {
    ZLINESPTR(0) = ZLINESPTR(1);
    ZLINESNR(0)  = ZLINESNR(1);
    restoreZlinenptr();
    actualEditor->wdy--;
  }
  actualEditor->ypos = ZLINESNR(actualEditor->wdy);

  printAll();
}
else
  if (l)
```

```
        {
          /* If fold marker:  redisplay window: */
          if (++actualEditor->wdy >= actualEditor->wch)
          {
            ZLINESPTR(0) = ZLINESPTR(1);
            ZLINESNR(0)  = ZLINESNR(1);
            restoreZlinenptr();
            actualEditor->wdy--;
          }
          actualEditor->ypos = ZLINESNR(actualEditor->wdy);

          printAll();
        }
        else
        {
          /* it was scrolled: */
          if (++actualEditor->wdy >= actualEditor->wch)
          {
            /* Cursor was on last line: */
            register UWORD nr;

            scrollUp((UWORD) 1);
            nr = (--actualEditor->wdy) - 1;
            actualEditor->ypos = ZLINESNR(actualEditor->wdy);
            printLine(ZLINESPTR(nr),actualEditor->yoff
                          + actualEditor->ch*nr);
          }
          else
          {
            /* Cursor near middle of window: */
            register UWORD nr;

            if (ZLINESPTR((nr = actualEditor->wdy) + 1))
            {
              /* move rest of window down: */
              ScrollRaster(actualEditor->rp,
                      0L,(LONG)-actualEditor->ch,
                      (LONG)actualEditor->xoff,
                      (LONG)actualEditor->yoff
                      + actualEditor->wdy*actualEditor->ch,
                      (LONG)actualEditor->xscr,
                        (LONG)actualEditor->yscr);
            }
            actualEditor->ypos = ZLINESNR(actualEditor->wdy);

            printLine(ZLINESPTR(nr),actualEditor->yoff
                          + actualEditor->ch*nr);
            nr--;
            printLine(ZLINESPTR(nr),actualEditor->yoff
                          + actualEditor->ch*nr);
          }
        }

      return (TRUE);
    }
    else
      return (FALSE);
  }
  else
  {
    cursorHome();
    cursorDown();
```

```
    return (FALSE);
  }
}

/*****************************
 *
 * deleteLine:
 *
 * delete line
 * that the cusor is on.
 * FALSE returned if
 * it cannot be deleted
 *
 *
 *****************************/

BOOL deleteLine()
{
  register struct Zline *line,*next,*prev;
  register UWORD l = 0;
  UWORD nextnr,prevnr;

  if (line = ZLINESPTR(actualEditor->wdy))
  {
    /* mark the deleted line: */
    if (line == actualEditor->actual)
    {
      actualEditor->actual = NULL;
      actualEditor->bufypos = 0;
      actualEditor->buflen  = 0;
    }

    actualEditor->changed = 1;
    actualEditor->num_lines--;

    /* Reconstruct folding: */
    if (line->flags & ZLF_FSE)
    {
      register struct Zline *z;

      /* Fold level for all following lines: */
      if ((z = line->succ)->succ)
      {
        if (((line->flags+1) & ZLF_FOLD) == (z->flags &
ZLF_FOLD))
          /* line was Fold start: */
          l = -1;
        else if ((line->flags & ZLF_FOLD) == ((z->flags+1) &
ZLF_FOLD))
          /* line was Fold end: */
          l = 1;
        else
          l = 0;

        if (l)
          while (z->succ)
          {
            z->flags = (z->flags & ~ZLF_FOLD)
                  | ((z->flags + l) & ZLF_FOLD);
            z = z->succ;
          }
```

```
        }
      }

      /* Next and previous line marked: */
      prevnr= ZLINESNR(actualEditor->wdy);
      prev  = prevLine(line,&prevnr);
      nextnr= ZLINESNR(actualEditor->wdy);
      next  = nextLine(line,&nextnr);

      /* Security check: */
      while ((prev == NULL) && (next == NULL) && (actualEditor-
>minfold))
      {
        l = 1;              /* Window completely output */
        actualEditor->minfold--;

        prevnr= ZLINESNR(actualEditor->wdy);
        prev  = prevLine(line,&prevnr);
        nextnr= ZLINESNR(actualEditor->wdy);
        next  = nextLine(line,&nextnr);

        if (prev || next) break;
        if (actualEditor->minfold) continue;

        if ((next = actualEditor->zlines.head)->succ == NULL)
          next = NULL;
        else
          nextnr = 1;

        break;
      }

      /* delete line */
      deleteZline(line);

      /* pointer to actual line set for  recalc: */
      if (next)
      {
        ZLINESPTR(actualEditor->wdy) = next;
        ZLINESNR(actualEditor->wdy)  = nextnr;
      }
      else
      {
        ZLINESPTR(actualEditor->wdy) = prev;
        ZLINESNR(actualEditor->wdy)  = prevnr;
      }

      /* Restore window contents: */
      if (l)
      {
        actualEditor->wdy = recalcTopOfWindow(actualEditor->wdy);
        restoreZlinenptr();
        printAll();
        cursorOnText();
      }
      else
      {
        /* with scrolling: */
        register UWORD oldwdy;

        oldwdy = actualEditor->wdy;
        actualEditor->wdy = recalcTopOfWindow(actualEditor->wdy);
```

```
        restoreZlinenptr();

        if (next)
        {
          /* push up rest of window: */
          ScrollRaster(actualEditor->rp,
                    0L, (LONG)actualEditor->ch,
                    (LONG)actualEditor->xoff,
                    (LONG)actualEditor->yoff
                    + actualEditor->wdy*actualEditor->ch,
                    (LONG)actualEditor->xscr,
                    (LONG)actualEditor->yscr);

          printLine(ZLINESPTR(actualEditor->wch - 1),
                 actualEditor->yoff
                 + actualEditor->ch*(actualEditor->wch - 1));
        }
        else
          if (oldwdy == actualEditor->wdy)
          {
            /* scroll top section of window down: */
            ScrollRaster(actualEditor->rp,
                      0L, (LONG)-actualEditor->ch,
                      (LONG)actualEditor->xoff,
                      (LONG)actualEditor->yoff,
                      (LONG)actualEditor->xscr,
                      (LONG)actualEditor->yoff
                      + (actualEditor->wdy + 1)
                       *actualEditor->ch - 1);

            printLine(ZLINESPTR(0),actualEditor->yoff);
          }
          else
            printLine(ZLINESPTR(oldwdy),actualEditor->yoff
                   + actualEditor->ch*oldwdy);

          cursorOnText();
        }

        return (TRUE);
      }
      else
        return (FALSE);
}
```

Now a few remarks about the functions:

- The character strings which represent the fold marks are defined as global variables at the beginning of the module. FSE_SIG specifies the number of characters needed for the fold mark; in this case `/*#FOLD:` represents the fold start and `/*#ENDFD` represents the fold end. This character string can be changed so that the editor can accomodate other programming languages (e.g., `;#FOLD:` could be used in assembler source). FSE_LEN supplies the character string's total length. This length must be identical for both starting marks and ending marks. The entire length will be needed later when we automate the omission of text segments. Then the editor must insert a line before and after

the fold mark. The user must remember to add the closing comment mark after the fold marks.

- The `getLineForEdit` function shows a change in the `convertLineForPrint` function. It returns the length of the converted line. The fifth parameter requires a pointer to a UBYTE variable in which `lastchar` is saved.

- Notice the new function named `cursorOnText`. It checks to see whether the cursor is in an existing line or if it was already moved past the end of text or fold. In the latter case the cursor moves to the last line in the window.

- The `deconvertLine` function performs the opposite function of the `Output` module's `convertLineForPrint` function. It checks each tab (tab = 0) for preceding spaces (fb! =0). If leading spaces exist, the function replaces these spaces with tabs.

- The `convertLineForPrint` function has yet another talent. When the program encounters a set `actualEditor->skipblanks` flag, it deletes all unnecessary spaces at the end of a line. If you try to edit text entered with a preset right margin less than the right border of the full size window (i.e., a line width less than 80 columns), you cannot delete the dots following the formatted line (more on this later).

- The `getFoldInc` function tests to see whether a line represents a fold mark, and returns zero if not. Otherwise `getFoldInc` returns a value of 1 for a fold start mark and -1 for an ending fold mark. These values are compatible with the `saveLine` function.

- The `saveLine` function controls setting the fold level if the line is a fold marker.

The following observations concern the above items. If a normal line is changed to a fold start mark, then the fold levels of all of the following lines must be incremented by one.

If a fold start mark is changed to a normal line and no other fold marks exist after this, fold levels of all of the following lines must be decremented.

These increment and decrement rules also apply to the fold end mark. If a line is a fold mark before as well as after a change, things can be complicated. However, creating a table of values, which returns the value by which all successive folds must be increased or decreased, makes things much simpler. This value depends on the status of a line (normal line or fold mark). The process works like this: The `getFoldInc` function handles a line changed to a fold mark and notes the return value. A fold mark changed to a normal line forces a

comparison between the fold level of the current line and the fold level of the next line. In this case we get the negative value relative to `getFoldInc` (-1 for the start and +1 for the end). Now we add the two values, and receive the value that must be added to the fold level of all subsequent lines.

So much for folding. The `changed` flag from the `Editor` structure determines whether something in the text has been changed. If the user changes a line with `saveLine`, the `changed` flag must be set.

The line is displayed at the end of the function.

- The `getBufferPointer` function helps convert or copy a line to the editor buffer. It initilaizes the three variables that simplify editing, a pointer to the character at which the cursor is found, the number of characters to the left of the cursor and the length of the rest of the line. If the cursor follows the last character of the line, the program inserts enough spaces to keep one space directly to the left of the cursor. The `insertChar` function inserts a character at the current cursor position.

- In overwrite mode, pressing the <Tab> key moves only the cursor to the right. If you move the cursor in this way, don't change text and don't type spaces over the text.

- The `insertLine` function prohibits division of a fold mark by pressing <Return>. This would create two new fold marks from the one fold mark. We chose not to solve this, but feel free to expand the program to work in this case. Instead you can enter a new fold mark and press <Return>, and the fold level change will remain harmless.

Text output in the window is another matter. If you insert a new line, the window contents are moved out of order. It's possible to use the `printAll` function, but that would be too slow. Besides, we want the editor to allow scrolling. The `insertLine` function should contain the following:

When the window contents must be scrolled horizontally, or when the newly insert line is a fold marker, `printAll` displays the window contents. This would not be necessary with horizontal scrolling, but if we scroll the window contents horizontally and then vertically, `printAll` is not much slower. Otherwise it scrolls vertically and the section of text at the cursor position scrolls down. When the cursor is on the bottom line, the entire contents of the window scroll up.

- Scrolling must also be enabled in the `deleteLine` function. The cursor remains in the current line and the rest of the text scrolls up. When the cursor is in the last line and `deleteLine` is invoked, the section above it scrolls down. If there is no text

in the top half of the window, the Y position of the cursor decrements.

Another problem arises in this function when the cursor is in a fold at the end of the text (`minfold` > 0), from which all lines are deleted. Then the editor cannot access the lines in this fold—this is why the `deleteZLine` function reduces `minfold` until it finds a line.

The fold marker can also be erased with `deleteZLine`. Then the fold level of all subsequent lines is changed accordingly.

The changes to the modules are fairly extensive, so let's begin with `Editor.h`. We removed the `toppos` variable from the `Editor` structure because the `zlinesnr` array already performs the same task. The following elements are new:

```
UBYTE buflastchar;
UWORD buflen,bufypos;
UWORD tabs       : 1;
UWORD skipblanks : 1;
UWORD autoindent : 1;
```

The `Output.c` module is next, into which we must add the `convertLineForPrint` function. Here's the altered function header:

```
UWORD convertLineForPrint(line,len,w,buf,lc)
UBYTE               *line;
register WORD              len;
register UWORD               w;
register UBYTE               *buf;
UBYTE                          *lc;
```

We need another local variable:

```
UWORD realLen;
```

These variables determine the length of the converted line. The following conversion remains almost as it was. However, we must specify the line length before the rest of the line fills with `lastchar`:

```
/* Save the length of the line */
realLen = l - 1;
```

Now the rest of the line fills with `lastchar` and the function returns the length:

```
*lc = lastchar;
return (realLen);
```

We made so many changes to the `printLine` function that we chose to reprint the function in its entirety below. This function displays the buffer if the ZLF_USED flag of the line was set. If the line does not exist (`line` = null), the corresponding window area is deleted (as in the `deleteLine` function):

```
void printLine        (line,y)
register struct Zline *line;
register UWORD             y;
{
  static UBYTE buf[MAXWIDTH];

  if (line)
  {
    register struct Zline *z;
    register UBYTE *p1,*p2;
    register UWORD w;
    register ULONG pen;
    UBYTE lc;

    /* check if line was processed: */
    if (line->flags & ZLF_USED)
    {
      /* copy buffer after buf, since printAt changed this! */
      if (MAXWIDTH > actualEditor->leftpos)
      {
        w = MAXWIDTH - actualEditor->leftpos;
        p2 = actualEditor->buffer + actualEditor->leftpos;

        p1 = buf;
        while (w)
        {
          *p1 = *p2;
          p1++;
          p2++;
          w--;
        }
      }
      else
        p1 = buf;

      /* fill rest with lastchar: */
      p2 = buf + (w = actualEditor->wcw);
      lc = actualEditor->buflastchar;
      while (p1 < p2)
      {
        *p1 = lc;
        p1++;
      }
    }
    else
      convertLineForPrint(line+1,line->len,
                    w = actualEditor->wcw,buf,&lc);

    /* If start/end marker of fold => FOLDPEN */
    if (line->flags & ZLF_FSE)
    {
      if ((z = line->succ)->succ)
      {
        if ((z->flags & ZLF_FOLD) <= actualEditor->maxfold)
        {
          SetAPen(actualEditor->rp,FOLDPEN);
          pen = FOLDPEN;
        }
        else
          pen = FGPEN;
      }
      else
```

```
        {
          SetAPen(actualEditor->rp,FOLDPEN);
          pen = FOLDPEN;
        }

        printAt(buf,w - SplitDec,actualEditor->xoff,y,pen);

        /* Old write color first again: */
        if (pen != FGPEN)
          SetAPen(actualEditor->rp,FGPEN);
      }
      else
        printAt(buf,w - SplitDec,actualEditor->xoff,y,FGPEN);
    }
    else
    {
      /* case no line => erase in window: */
      struct RastPort *rp = actualEditor->rp;

      SetAPen(rp,BGPEN);
      RectFill(rp,(ULONG)actualEditor->xoff,
               (ULONG)y,
               (ULONG)actualEditor->xscr,
               (ULONG)y + actualEditor->ch - 1);
      SetAPen(rp,FGPEN);
    }
}
```

The `Cursor.c` module has also undergone some changes. The `recalcTopOfWindow` function is new. This function determines the line position of the top line of the window and places the value in the `linesptr` and `linesnr` arrays. This function initializes the `linesptr` array after a text alteration, or after changing `minfold` or `maxfold`. The cursor remains at the old position.

```
UWORD recalcTopOfWindow(y)
register UWORD          y;
{
  register struct Zline *z,*zp;
  register UWORD znrp;
  UWORD znr,oldy = y;

  z  = ZLINESPTR(y);
  znr = ZLINESNR(y);
  while (y)
  {
    zp    = z;
    znrp  = znr;
    if ((z = prevLine(z,&znr)) == NULL)
      break;
    y--;
  }

  if (y)
  {
    ZLINESPTR(0) = zp;
    ZLINESNR(0)  = znrp;
  }
  else
  {
```

```
    ZLINESPTR(0) = z;
    ZLINESNR(0)  = znr;
  }

  return (oldy - y);
}
```

This function keeps the cursor in the current line when encountering a fold and when deleting a line. The new line position is returned from this function, which usually matches the old line position. Now when encountering a fold, the cursor can be relatively far down in the window, but not very much text is visible. Then the cursor must be moved up where the new position accesses this function.

The program segment for fold handling is placed before the `handleKeys` function:

```
/*************************
 *
 * enterFold:
 *
 * STEP one fold in.
 *
 *************************/

void enterFold()
{
  register struct Zline *z;

  if (actualEditor->maxfold < ZLF_FOLD)
  {
    actualEditor->maxfold++;
    if (z = ZLINESPTR(actualEditor->wdy))
      if ((z = z->succ)->succ)
        /* If there is a following line: */
        if (((ZLINESPTR(actualEditor->wdy)->flags & ZLF_FOLD)
            < (z->flags & ZLF_FOLD))
          && ((z->flags & ZLF_FOLD) == actualEditor->maxfold))
        {
          /* When this begins a new fold: */
          actualEditor->minfold = actualEditor->maxfold;
          ZLINESPTR(0)   = z;
          ZLINESNR(0)    = ZLINESNR(actualEditor->wdy) + 1;
          actualEditor->wdy = 0;
        }

    actualEditor->wdy = recalcTopOfWindow(actualEditor->wdy);
    restoreZlinenptr();
    printAll();
    actualEditor->ypos = ZLINESNR(actualEditor->wdy);
  }
}

/*****************************
 *
 * exitFold:
 *
 * Step one fold out.
 *
 *****************************/
```

```
void exitFold()
{
  register UWORD wdy;

  if (actualEditor->maxfold)
  {
    actualEditor->maxfold--;
    if (actualEditor->minfold > actualEditor->maxfold)
      actualEditor->minfold = actualEditor->maxfold;

    wdy = actualEditor->wdy;
    if (ZLINESPTR(wdy) == NULL)
      actualEditor->wdy = wdy = 0;

    if (ZLINESPTR(wdy))
      if (((ZLINESPTR(wdy)->flags & ZLF_FOLD) > actualEditor-
>maxfold)
      || ((ZLINESPTR(wdy)->flags & ZLF_FOLD) < actualEditor-
>minfold))
        ZLINESPTR(wdy) =
prevLine(ZLINESPTR(wdy),&(ZLINESNR(wdy)));

    actualEditor->wdy = recalcTopOfWindow(wdy);
    restoreZlinenptr();
    printAll();
    cursorOnText();
  }
}
```

The last function shows how the `recalcTopOfWindow` function is inserted. In addition to the existing functions for cursor movement, an additional function is needed for placing the cursor at the beginning of a line, based on the function used in `insertLine`:

```
BOOL cursorHome()
{
  actualEditor->xpos = 1;
  if (actualEditor->leftpos)
    scrollRight(actualEditor->leftpos);

  return (TRUE);
}
```

The following new `defines` are necessary to this version of the editor:

```
#define CHOME 'A'
#define UNDO '?'
#define CFOLD 6
#define CENDF 5
#define TABMODE 20
#define WRITEMODE 15
#define AUTOINDENT 1
#define DELLINE 2
#define DEL 127
#define BS 8
#define LF 10
#define CR 13
```

Examine the following listing for `handleKeys`, and you'll notice some changes:

```
void handleKeys(buf,len)
register UBYTE  *buf;
register WORD     len;
{
  register UBYTE first;

  while (len > 0)
  {
    first = *buf;
    buf++;
    len--;
    if ((first == CSI) && (len > 0))
    {
      len--;
      switch (*buf++)
      {
        case CUU:
          cursorUp();
          break;
        case CUD:
          cursorDown();
          break;
        case CUF:
          cursorRight();
          break;
        case CUB:
          cursorLeft();
          break;
        case SU:
          halfPageDown();
          break;
        case SD:
          halfPageUp();
          break;
        case ' ':
          if (len > 0)
          {
            len--;
            switch (*buf++)
            {
              case CHOME:
                cursorHome();
                break;
              default:
                if (! insertChar((UBYTE)CSI))
                  DisplayBeep(NULL);
                buf -= 2;
                len += 2;
            }
            break;
          }
          else
            goto noCSI;
        case UNDO:
          if ((len > 0) && (*buf == '~'))
          {
            buf++;
            len--;
```

```
            undoLine();
            break;
          }
          /* Else: default */
        default:
noCSI:
          /* No command sequence! */
          if (! insertChar((UBYTE)CSI))
            DisplayBeep(NULL);

          buf--;
          len++;
          break;
      } /* switch (first) */
    }
    else
      switch (first)
      {
        case CFOLD:
          enterFold();
          break;
        case CENDF:
          exitFold();
          break;
        case TABMODE:
          actualEditor->tabs = !actualEditor->tabs;
          printAll();
          break;
        case WRITEMODE:
          actualEditor->insert = !actualEditor->insert;
          break;
        case AUTOINDENT:
          actualEditor->autoindent= !actualEditor->autoindent;
          break;
        case DELLINE:
          if (! deleteLine())
            DisplayBeep(NULL);
          break;
        case DEL:
          if (! deleteChar())
            DisplayBeep(NULL);
          break;
        case BS:
          if (! backspaceChar())
            DisplayBeep(NULL);
          break;
        case LF:
        case CR:
          if (! insertLine(first))
            DisplayBeep(NULL);
          break;
        default:
          if (! insertChar(first))
            DisplayBeep(NULL);

      }   /* switch (first) */
  }     /* while (len) */
}
```

The first `switch case` instruction handles the CSI sequences as before. It is inconvenient that these CSI sequences do not have a set

length, some are two and three characters long. This instruction provides another `switch case` instruction which processes all of the sequences that begin with CSI ' '. This is currently only CSI ' ' 'A', which is sent through <Shift><Cursor Left>. This could be expanded later to include user defined sequences.

The second `switch case` instruction handles control characters that call their own functions at that moment. Later we will access a section of the command line.

Let's look at an overview of editor commands available so far:

Keys	Comments
Cursor keys	moves the cursor by one character/line
Shift Cursor up/down	scrolls text a half page up or down
Shift Cursor left	sets the cursor at the beginning of the line
Help	undo
Ctrl A	turns Auto Indent on/off
Ctrl B	erases the line the cursor is in
Ctrl E	leaves a fold
Ctrl F	enters a fold
Ctrl O	switches between insert and overwrite mode
Ctrl T	turns tabs on/off
Delete	as usual
Backspace	as usual

The functions from the new `Edit` module must be declared:

```
void undoLine();
void cursorOnText();
BOOL insertChar(),deleteChar(),backspaceChar(),
                                insertLine(),deleteLine();
```

The `deleteZline` function in the `Memory.c` module has been changed for two reasons:

- The contents of the `zlinesptr` and `zlinesnr` arrays remain unchanged
- The line is not erased from the list of the lines.

We have also eliminated the following lines following `Remove(line)` to keep `deleteZLine` fully operational:

```
/* remove line from list: (keep these two line)*/
Remove(line);

/* set pointer to this line to zero:  (delete these)*/
if (actualEditor->actual == line)
  actualEditor->actual = NULL;
for (n = 0, zptr = actualEditor->zlinesptr;
    n <= actualEditor->wch; n++, zptr++)
  if (*zptr == line)
  {
```

```
        *zptr = NULL;
        break;
    }
```

A few changes still need to be made to the main `Editor.c` module. The additional elements of the `Editor` structure must be initialized in the `OpenEditor` function:

```
ed->buflen     = 0;
ed->bufypos    = 0;
ed->buflastchar= ' ';
ed->tabs       = 1;
ed->skipblanks = 1;
ed->autoindent = 1;
```

It may surprise you when testing the cursor that the cursor runs on (keeps going after you release the cursor key) horizontally. This happens with almost all the functions that move the cursor or insert text, and when you set the keyboard repeat to very high speed from Preferences.

This effect can be prevented by checking to see if something has changed after the program processes a keypress. If this is the case, and if all of the `RAWKEY` messages have set the `REPEAT` flag, the first message is removed. The `REPEAT` flag is set from the operating system when a `RAWKEY` message is encountered instead of a keyboard repeat, because the user has pressed the key. Messages that do not set the `REPEAT` flag are not deleted because the user may sometimes press keys faster than the editor can process them.

The current messages are deleted until only one remains. This is processed during the next execution of the main loop. In principle we could also delete the first message, but then the cursor movement would be uneven. The fact that the cursor moves one character farther than you may want it to move is the price paid for the fast and smooth cursor movement.

The `RAWKEY` check looks like the following:

```
case RAWKEY:
{
  register struct IntuiMessage *im1,*im2;

  if (! (code & IECODE_UP_PREFIX))
  {
    inputEvent.ie_Code     = code;
    inputEvent.ie_Qualifier = qualifier;
    if ((inputLen = RawKeyConvert(
          &inputEvent,inputBuffer,MAXINPUTLEN,NULL)
          ) >= 0)
      handleKeys(inputBuffer,inputLen);
  }

  /* prevent running: */
  im1 = (struct IntuiMessage *)
      edUserPort->mp_MsgList.lh_Head;
```

```
    while (im2 = (struct IntuiMessage *)
            im1->ExecMessage.mn_Node.ln_Succ)
    {
      if (im2->ExecMessage.mn_Node.ln_Succ == NULL) break;
      if (im1->Class != RAWKEY) break;
      if (!(im1->Qualifier & IEQUALIFIER_REPEAT)) break;
      if (im2->Class != RAWKEY) break;
      if (!(im2->Qualifier & IEQUALIFIER_REPEAT)) break;

      /* Message removed: */
      im1 = GetMsg(edUserPort);
      ReplyMsg(im1);

      im1 = (struct IntuiMessage *)
          edUserPort->mp_MsgList.lh_Head;
    }

    break;
  }
```

The following function must be called directly after the `switch case` assignment:

```
saveIfCursorMoved();
```

This function must be declared as `void` in the external function list.

To compile this version, look for the name of the object file `edit.o` in the `makefile`, and add the following line to the end of the `makefile`:

```
src/Edit.o: src/Edit.c src/Editor.h /pre/Editor.pre
```

The optional diskette for the book contains the source text in the `v0.5` directory. After you compile the new program you have quite a bit to do because there are many options to test. Start the program using the `TestText` file as described in the last section. An error that we found in the program during development has been left in the program for you. You should be able to find it for it yourself.

Debugging

Have you found it? If you want to add characters to the end of a line in the overwrite mode, the editor does not enter the new characters when you exit the line again. We looked over the source text and couldn't find an error anywhere. This is where the debugger helps us. Compile the editor using `make debug`. Start the debugger with `db` and then the editor, as we have already described (your drive setup may differ):

```
sys3:bin/db
Editor <TestText
```

The debugger displays the editor as usual. Now think about how to proceed further. The error occurs when we want to add a character at the end of the line using the `insertChar` function. Set a breakpoint at this function:

```
bs insertChar
```

and start the editor by pressing <g> (Go). Activate the editor window and press <Ctrl><O> to enter overwrite mode. Search for the end of a line, or for a blank line. Move the cursor there and press a key (e.g., <a>). The editor stops and the debugger displays the beginning of the `insertChar` function. Now you need to find the error! The reason could be that the length of the buffer was not increased correctly when you want to add a character. It would be best to move through this function in single step mode.

First the `LINK` and `MOVEM` instructions perform general initialization. Then register `D4` receives the inserted character and treats the character as a register variable. The local variable `-a(A%)` is filled with the value one, which it handles as `xadd`. The `getBufferPointer` function call occurs. Press <t> to bypass this function, not <s>. Pressing <t> executes a command completely and executes a subroutine of `BSR` or `JSR` with a subroutine call, before program control returns to the debugger.

Now we look at the values written in the `ptr`, `inc` and `rest` variables:

```
p2dx a5-8
```

Before the `getBufferPointer` function call, three addresses are placed on the stack, which are the addresses of the three variables sought:

```
-8(a5)
-6(a5)
-4(a5)
```

Because variables in C always appear on the stack in the order in which they were defined, we can assume the following association:

```
rest = -8(a5)
inc = -6(a5)
ptr = -4(a5)
```

To display this we must display two words at address `-8(a5)` and then a long word, which the debugger calls `p2dx`. In addition, two decimal and one hexadecimal long word must appear from the start address. The values for `inc` and `rest` are both zero if the cursor moves to an empty line. The value of `ptr` is not of interest, but look where `ptr` points:

```
db *(a5-4)
```

The line in the debugger window consists of an address, and equal sign, and a 20. `ptr` points to a string that is made up only of spaces (`$20`). Let the debugger continue in single step mode until everything is correct. It follows the testing of the return values and then checks to see if D4 contains the value 9. D4 is the inserted character, and 9

corresponds to a tab. Because you pressed a normal letter, the program continues.

A value passes from the `Editor` structure and the first bit is marked from there and tested. If this bit equals zero, the editor is in overwrite mode and the program continues. When you disassemble the program from here, you should see the following:

```
_insertChar+184    movea.l -4(a5),a0
_insertChar+188    move.b  d4,(a0)
_insertChar+18a    clr.w   -8(a5)
_insertChar+18e    beq.s   _insertChar+19e
_insertChar+190    movea.l _actualEditor,a0
_insertChar+194    addq.w  #1,76(a0)
_insertChar+198    move.w  #1,-8(a5)
_insertChar+19e
```

This short program segment corresponds to the following lines which allow character insertion in overwrite mode. These lines appear at the end of the `insertChar` function:

```
*ptr = c;
if (rest = 0)
{
   actualEditor->buflen++;
   rest = 1;
}
```

The characters at the location indicated by `ptr` are saved. The `rest` variable should be set to zero (instead of memory location `-8(A5)`). Here's the error: The single equal sign between `rest` and `0` in the second line. That line should appear as follows:

```
(rest == 0);
```

Correct the error and recompile the program if you wish.

C's ability to write instructions appears here as a hindrance to programmers. However, this same ability lets the programmer create intricate functions.

Local variables

Before going on, we want to discuss a difficult subject with you—local variables. You already know that a block is a set of functions and statements which are enclosed in a pair of curly braces. Local variable definitions usually appear at the beginning of a block. So far in this book, local variables have been used only in conjunction with functions in programs. Local variables can be defined at the beginning of a function. In fact, you can define local variables in any block.

Local variables help clarify the variable structure, provided the variables are not treated as register variables. Register variables cause problems because of the limited number of accessible registers (in controllers five to seven). This can be fixed by placing local variables in blocks: Define register variables in just the block that requires them, and place all of

the necessary instructions in a separate block. This releases the corresponding registers outside the block, and can be handled by the C compiler in another way. Since more register variables can be used, the finished program runs faster. Look at the `slow` and `fast` functions listed below:

```
slow()
{
   register int a,b,c;
   register int a1,b1,c1;
   register int a2,b2,c2;
   for (a1 = a, b1 = b, c1 = c; a1 > 0; a1--)
      ;
   for (a2 = a, b2 = b, c2 = c; a2 > 0; a2--)
      ;
}
fast()
{
   register int a,b,c;
   {
      register int a1,b1,c1;
      for (a1 = a, b1 = b, c1 = c; a1 > 0; a1--)
         ;
   }
   {
      register int a2,b2,c2;
      for (a2 = a, b2 = b, c2 = c; a2 > 0; a2--)
         ;
   }
}
```

These serve the same purpose. The difference is in the use and placement of register variables. Compile each to get the following assembler source texts:

```
                        ;slow()
                        ;{
    public _slow
_slow:
    link a5,#.2
    movem.l .3,-(sp)
                        ; register int a,b,c;
                        ; register int a1,b1,c1;
                        ; register int a2,b2,c2;
                    ;
                    ; for (a1 = a, b1 = b, c1 = c; a1 > 0; a1-
-)
    move.w d4,d7
    move.w d5,a2
    move.w d6,a3
    bra .7
.6
                        ; ;
.4
    sub.w #1,d7
.7
    tst.w d7
    bgt .6
.5
                    ;
```

```
                        ; for (a2 = a, b2 = b, c2 = c; a2 > 0; a2--
)
     move.w d4,-2(a5)
     move.w d5,-4(a5)
     move.w d6,-6(a5)
     bra .11
.10
                              ;  ;
.8
     sub.w #1,-2(a5)
.11
     tst.w -2(a5)
     bgt .10
.9
                              ;
                              ;}
.12
     movem.l (sp)+,.3
     unlk a5
     rts
.2   equ -6
.3   reg d4/d5/d6/d7/a2/a3
                              ;
                              ;fast()
                              ;{
     public _fast
_fast:
     link a5,#.13
     movem.l .14,-(sp)
                             ; register int a,b,c;
                             ;
                        ; {
                        ;  register int a1,b1,c1;
                        ;
                        ;  for (a1 = a, b1 = b, c1 = c; a1 > 0; a1-
-)
     move.w d4,d7
     move.w d5,a2
     move.w d6,a3
     bra .18
.17
                              ;   ;
.15
     sub.w #1,d7
.18
     tst.w d7
     bgt .17
.16
                        ; }
                        ;
                        ; {
                        ;  register int a2,b2,c2;
                        ;
                        ;  for (a2 = a, b2 = b, c2 = c; a2 > 0; a2--
)
     move.w d4,d7
     move.w d5,a2
     move.w d6,a3
     bra .22
.21
                              ;   ;
.19
```

```
    sub.w #1,d7
.22
    tst.w d7
    bgt .21
.20
                            ; }
                            ;
                            ;}
.23
    movem.l (sp)+,.14
    unlk a5
    rts
.13 equ 0
.14 reg d4/d5/d6/d7/a2/a3
    dseg
    end
```

These instructions clearly show which variables are register variables and which aren't. The first `for` loop of the `slow` function has more obvious registers, but the second `for` loop writes the register values into memory. This loop defines too many register variables, which means that the C compiler must convert some of them into normal variables. The `fast` function uses register variables exclusively, which are all available when the function executes.

The coming functions require local variable definitions within blocks for inclusion in the editor source. You may want to experiment with local variables on your own. They can often help make a program faster while keeping the source text readable, without resorting to assembler routines.

4.3.9 Command Line

File access must still be created for our editor to be complete. Since filenames must be entered, a command line should be included in the editor, like the ED editor program. In addition, a status line should exist, which displays the current cursor position and other information.

On to the command line. This should serve a purpose similar to the one in ED: The command line should be accessible by pressing the <Esc> key, and should accept two-letter commands. ED's command set isn't configured in an easy to remember order. For example, look at ED's block commands: BS marks the start of a block, but you must enter IB to insert a block, instead of BI.

Our own editor should have commands which have a first letter indicating the general command grouping. Let's consider which command groups would be needed by an editor. First priority would be

loading and saving data. We also need commands for cursor control, text block manipulation, setting mode flags, text deletion, word searching and exiting the program. Also, commands may be needed if you choose to extend the editor to a programmable mode. Here are a few commands we came up with by brainstorming (the ones marked with an asterisk are implemented in our version of the editor):

ASCII file commands	`AF ["Filename"]`	saves text within a folded section	
	`AL ["Filename"]`	loads an ASCII text file	*
	`AS ["Filename"]`	saves the entire text	*
Block commands	`BC (Block copy)`	copies block to buffer	
	`BD (Block delete)`	deletes block	
	`BE (Block end)`	marks end of block	
	`BF (Block fold)`	folds block	
	`BH (Block hide)`	deletes block marker	
	`BM (Block move)`	moves block	
	`BP (Block paste)`	inserts block contained in buffer	
	`BS (Block start)`	marks beginning of block	
	`BX (Block cut)`	deletes block from text to buffer	
Cursor commands	`CB (Cursor bottom)`	moves cursor to end of text/fold	*
	`CD (Cursor down)`	moves cursor one line down	*
	`CE (Cursor end)`	moves cursor to end of line	*
	`CH (Cursor home)`	moves cursor to start of line	*
	`CL (Cursor left)`	moves cursor one character left	*
	`CN (Cursor next)`	moves cursor to start of next line	*
	`CP (Cursor previous)`	moves cursor, start of previous line	*
	`CR (Cursor right)`	moves cursor one character right	*
	`CS (Cursor start)`	moves cursor to start of line	*
	`CT (Cursor top)`	moves cursor to start of text/fold	*
	`CU (Cursor up)`	moves cursor one line up	*
Delete commands	`DB (delete back)`	backspace	*
	`DC (delete char)`	delete character	*
	`DL (delete line)`	delete line	*
Exchange commands	`EN ["String1"String2"]`	replaces next occurrence	
	`EP ["String1"String2"]`	replaces previous occurrence	
Find commands	`FN ["String"]`	searches for next occurrence of string	
	`FP ["String"]`	searches backwards	
IF command	`IF (condition) assigment`		

Line commands	`LA "String"`	inserts string as new line behind current line in text (append)	
	`LI "String"`	inserts string as new line before current line in text (insert)	
	`LJ (line join)`	adds one line to following line	
	`LS (line split)`	splits line at cursor position	
Macro command	`M[A-Z][=instructions]`	execute or set macro	
Quit command	`Q`	exits editor	*
Repeat command	`RP instruction`	repeats instruction until stopped	
Set commands	`Sa`	Auto Indent off	*
	`SA`	Auto Indent on	*
	`Sb`	SkipBlanks off	*
	`SB`	SkipBlanks on	*
	`Si`	insert mode	*
	`So`	overwrite mode	*
	`St`	tabulators off	*
	`ST`	tabulators on	*
Tab commands	`TC (Tab clear)`	erases tab at cursor position	
	`TS (Tab set)`	sets tab at cursor position	
eXit command	`X`	saves changed text and exits	*
Undo command	`U`	undo changes in current line	

The editor commands, with the exception of the `Set` commands, accept commands entered in either uppercase or lowercase.

This section lists the development process of the most important editor features, such as saving and loading text. Some of the features already exist from the previous versions (e.g., cursor movement).

The first item needed is a command line that works independently from the normal editing functions. A string gadget placed at the lower left border of the window serves this purpose. Use of a string gadget means that you can enter text in the gadget immediately, without clicking on it to activate it. The string gadget becomes active the moment the user presses the <Esc> key. The main difference between our command line and `ED`'s <Esc> commands is that you can edit the text in the string gadget.

The editor must execute the commands of the command line as soon as the user presses the <Return> key. The `RELVERIFY` flag indicates a pressed <Return> key to the editor. The editor then receives a

GADGETUP message each time the user presses <Return> while in a string gadget. It must then determine which command was entered and execute the appropriate function. Because we have a complete range of commands, we could have the editor compare the command line with all known commands. There is no need to do this, since our commands are sorted according to group, and all commands in one group begin with the same letter. The editor reads the first character, moves to that command group, then executes the function called by the second character.

A pointer to the second letter of the command is given in this function because the first is already known. If the function can execute the command, this pointer returns to the character after the command, otherwise it returns an error message. The `executeCommand` function calls these commands. This function skips spaces and looks for semicolons (which allow the execution of multiple commands in one line).

The command line also allows the user to enter multiple commands enclosed in curly braces, with a number preceding the first brace. The editor executes the commands within braces the number of times indicated by the preceding number. The following command sequence moves the current line and the nine lines after it three characters to the left:

```
CS;10{3DC;CD};10CU
```

The cursor then returns to the original line. Two items of interest in this command:

1.) The `saveIfCursorMoved` function must be called because the cursor may have been moved.

2.) The user must be able to stop a command in process without losing data. Command execution stops when the user presses any key.

Info line

The info line appears along the bottom border of the window, to the right of the command line. This line actually contains a minimum amount of information, to keep the info line small and maximize the size of the text area:

- X and Y position of the cursor
- Number of lines in the text
- `minfold` status
- `maxfold` status
- Flag status

If we limit the maximum width for each number to four digits (two digits for `minfold` or `maxfold`), the maximum is 34 characters. The info line looks something like this:

```
X=nnnn Y=nnnn #=nnnn [nn,nn] ICATB
```

The X and Y indicate the cursor's present coordinates. The # represents the total number of lines. The first number in brackets display `minfold`, and the second number displays `maxfold`. The five characters at the end of the info line represent the flags:

I:	insert / O:Overwrite
C:	changed.
A:	autoindent.
T:	tabs.
B:	skipblanks.

Lower case characters indicate a disabled flag, while upper case characters indicate an enabled flag. We set the width of the command line relative to the window's width. This avoids any overlapping between the info line and the command line if the user reduces the size of the window. The minimum width of the window must be set accordingly—the command line cannot have a negative width if you make the window too small.

Assembler and C code

Functions in the editor display a value or group of values in the info line without constantly redisplaying the info line. Because speed is important for displaying the cursor position, this is a good opportunity to write a subroutine in assembler and append it to our program. To keep the assembler programming to a minimum, we'll tell the program to convert a number to a string in assembler programming.

The C library's `ftoa` function performs this task, but `ftoa` requires a floating point number. The C compiler converts the integer number into a floating point number, then converts the floating point number to the necessary string. Our function converts an integer number directly, where we can also determine the length of the character string.

The Aztec C compiler allows easy combination of assembler and C code. Instead of the function header we first place the `#asm` compiler instruction at the beginning of a line. Then we enter our assembler program and end it with an `#endasm` instruction. The compiler now knows that an assembler program starts after `#asm` and ends before `#endasm`. The compiler initially bypasses this assembler code. The assembler, which is called from the C compiler, compiles this code. The assembler's routine address must be indicated so that the assembled code may be accessed from C functions. If you want the routine accessed under the default name `cn_utoa` (CoNvert Unsigned To

Ascii), you must include the following line at the beginning of the routine:

```
global _cn_utoa
_cn_utoa:
```

The `global` statement defines the label `_cn_utoa`. This label can now be accessed like a global variable or a function. The underscore character preceding the function name is necessary because the C compiler precedes all of the labels with an underscore when compiling a C program into an assembler program. If you have already seen what the C compiler creates for assembler programs, you have probably noticed this.

The parameters that are defined first are placed on top of the stack. The following function call:

```
cn_utoa(string,value,length);
```

returns parameters as the following addresses on the stack if value and length are of type `UWORD`:

```
0(SP) ^ return address
4(SP) ^ string
8(SP) = value
10(SP) = length
```

We cannot use all of the free registers in our assembler program. The registers that the C compiler uses for reference to local (A5) and global (A4) variables may be changed, like the register that uses register variables (D4-D7, A2, A3). However, there are more than enough registers for us, especially since we only need four segments for our subroutine. Here is the complete assembler program, which will be located in `Output.c`:

```
/*****************************
 *
 * cn_utoa(buf,val,len):
 *
 * Convert UWORD in ASCII.
 *
 * buf ^ buffer for ASCII.
 * val = Value
 * len = Maximum Lenght.
 *
 *****************************/

void cn_utoa();

#asm

  cseg
  public _cn_utoa
```

```
_cn_utoa:

  ; 4(sp) ^ Buffer,
  ; 8(sp) = Value,
  ;10(sp) = maximum Lenght.

  lea     4(sp),a0
  move.l  (a0)+,a1
  moveq   #0,d0
  move.w  (a0)+,d0       ; = Value
  move.w  (a0),d1      ; = maximum Lenght
  lea     0(a1,d1.w),a0 ; ^ End of Buffers
  bra.s   .in

.lp:
  divu    #10,d0
  swap    d0
  add.b   #'0',d0 ; d0.w = VAlue MOD 10
  move.b  d0,-(a0)
  clr.w   d0
  swap    d0     ; Zero-Flag is 1, when d0.l = 0
.in:
  dbeq    d1,.lp     ; length is value = 0
          ;or String full
  beq.s     .ib    ; Value = 0 => Rest is
        ;full of spaces
  bra.s   .r  ; String full!

.bl:
  move.b  #' ',-(a0)
.ib:
  dbra    d1,.bl
.r:
  rts

#endasm
```

The program fills in the string starting from the end. The number to be converted divides by ten each time, and the remainder from the division is written in the string as numerals. The rest of the string fills with blanks, as long as memory is available.

The values must be displayed properly in the info line. We need the position at which this display must begin. Add to that the position of the gadget and the position of the `IntuiText` structure (which the info line contains). The `Intuitext` structure determines the character set which keeps the info line as small as possible We recommend `Topaz_80`, which permits 80 characters per line. We must also choose this character set for the output of the values. The `PrintIText` function allows character set assignment in the `IntuiText` structure. This function also requires a null byte at the end of the string. It would be better to use the text function from the graphics library, because you can enter the length of the character string. Consider that the value must be copied to the info line string, because the border is also redisplayed when the window is redisplayed. If we display the new value over the old value using `PrintIText` or

`Text` in the window, the new output appears the same as the old because it is always found in the string.

We want to copy or convert the new value (`xpos`, `ypos`, etc.) to the info line, then use `Text` to display the corresponding section of the info line. For this we must first set the character set with `SetFont`. Then `OpenFont` must open the character set for access by the main program. The value display functions look like the following (the assembler program `_cn_utoa` should precede these functions in the `Output.c` module!):

```
/*#FOLD: printXpos */
/*****************************
 *
 * printXpos:
 *
 * Give Xpos of cursor.
 *
 *****************************/

void printXpos()
{
  register struct TextFont *oldFont;
  register struct Editor *ae = actualEditor;

  cn_utoa(ae->gi_Text + INFO_XPOS_INC,ae-
>xpos,(UWORD)INFO_XPOS_LEN);

  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_xpos,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text + INFO_XPOS_INC,(ULONG)INFO_XPOS_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
/*#FOLD: printYpos */
/*****************************
 *
 * printYpos:
 *
 * Give Ypos of cursor.
 *
 *****************************/

void printYpos()
{
  register struct TextFont *oldFont;
  register struct Editor *ae = actualEditor;

  cn_utoa(ae->gi_Text + INFO_YPOS_INC,ae-
>ypos,(UWORD)INFO_YPOS_LEN);

  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_ypos,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text + INFO_YPOS_INC,(ULONG)INFO_YPOS_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
```

```
/*#FOLD: printNumLines */
/*****************************
 *
 * printNumLines:
 *
 * Give number of lines
 * of text
 *
 *****************************/

void printNumLines()
{
  register struct TextFont *oldFont;
  register struct Editor *ae = actualEditor;

  cn_utoa(ae->gi_Text + INFO_NUMOFLINES_INC,ae->num_lines,
        (UWORD)INFO_NUMOFLINES_LEN);

  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_numzlines,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text + INFO_NUMOFLINES_INC,
      (ULONG)INFO_NUMOFLINES_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
/*#FOLD: printFold */
/*******************************
 *
 * printFold:
 *
 * Gives  minfold and maxfold display.
 *
 *******************************/

void printFold()
{
  register struct TextFont *oldFont;
  register struct Editor *ae = actualEditor;

  cn_utoa(ae->gi_Text + INFO_MINFOLD_INC,ae->minfold,
        (UWORD)INFO_MINFOLD_LEN);
  cn_utoa(ae->gi_Text + INFO_MAXFOLD_INC,ae->maxfold,
        (UWORD)INFO_MAXFOLD_LEN);

  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_minfold,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text +
INFO_MINFOLD_INC,(ULONG)INFO_MINFOLD_LEN);
  Move(ae->rp,ae->ip_maxfold,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text +
INFO_MAXFOLD_INC,(ULONG)INFO_MAXFOLD_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
/*#FOLD: printFlags */
/*****************************
 *
 * printFlags:
 *
 * At Ypos of cursor.
```

```
 *
 ****************************/

void printFlags()
{
  register struct TextFont *oldFont;
  register struct Editor *ae = actualEditor;
  register UBYTE *fs = ae->gi_Text + INFO_FLAGS_INC;

  if (ae->insert)
    *fs++ = 'I';
  else
    *fs++ = 'O';
  if (ae->changed)
    *fs++ = 'C';
  else
    *fs++ = 'c';
  if (ae->autoindent)
    *fs++ = 'A';
  else
    *fs++ = 'a';
  if (ae->tabs)
    *fs++ = 'T';
  else
    *fs++ = 't';
  if (ae->skipblanks)
    *fs = 'B';
  else
    *fs = 'b';

  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_flags,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text +
INFO_FLAGS_INC,(ULONG)INFO_FLAGS_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
/*#FOLD: printInfo */
/****************************
 *
 * printInfo:
 *
 * Gives new  Infoline.
 *
 ****************************/

void printInfo()
{
  printXpos();
  printYpos();
  printNumLines();
  printFold();
  printFlags();
}
/*#ENDFD*/
```

All of these functions operate in the same manner: First the new value is copied into the string, the character set is changed and the changed section of the info line is redisplayed. The normal character set returns. The `defines` compute the positions of the individual values, and

form a new pair of elements in the `Editor` structure. The `initWindowSize` function initializes these new elements, which contain the pixel positions for all of the values. The new defines are defined in `editor.h`, so we must declare the new `SetFont` function in the `Output` module:

```
void SetFont();
```

We also need a pointer to the character set used for the info line, and which is defined and enabled in the main module:

```
extern struct TextFont *infoFont;
```

The positions of each value in the info line must be calculated in the `initWindowSize` function. This happens before restoration of the `linesptr` array:

```
{
  /* Position for Infoline: */
  register ULONG xinc,xsize;

  xinc = w->Width + ed->G_Info.LeftEdge + ed->gi_IText.LeftEdge
- 1;
  xsize= infoFont->tf_XSize;

ed->ip_xpos = xinc + INFO_XPOS_INC * xsize;
ed->ip_ypos = xinc + INFO_YPOS_INC * xsize;
ed->ip_numzlines = xinc + INFO_NUMOFLINES_INC * xsize;
ed->ip_minfold = xinc + INFO_MINFOLD_INC * xsize;
ed->ip_maxfold = xinc + INFO_MAXFOLD_INC * xsize;
ed->ip_flags  = xinc + INFO_FLAGS_INC * xsize;
ed->ip_topedge = w->Height + ed->G_Info.TopEdge
    + ed->gi_IText.TopEdge
    + (ULONG)infoFont->tf_Baseline - 1;
}
```

This concludes the changes made to the `Output.c` module. The `Editor.c` module defines the gadgets needed for the info and command lines. Now let's look at the source text for the `Command.c` module:

```
<src/command.c>

/************************************
 *
 *  Module: Command
 *
 *  Command-Interpreter for
 *  Editor.
 ************************************/

/*#FOLD:  Includes  */
/************
 *
 * Includes:
 *
```

```
 ************/

#include <stdio.h>
#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"
/*#ENDFD*/
/*#FOLD:  Defines */
/***********
 *
 * Defines:
 *
 ***********/

#define TAB 9
#define LF 10
#define CR 13

#define UC(c) (((c >= 'a') && (c <= 'z'))? (c & 0xDF): c)
#define DIGIT(c) ((c >= '0') && (c <= '9'))
#define SKIPBLANKS(str) while (*str == ' ') str++;
/*#ENDFD*/
/*#FOLD:  External Functions */
/**********************
 *
 * External Functions:
 *
 **********************/

BOOL
cursorLeft(),cursorRight(),cursorUp(),cursorDown(),cursorHome();
BOOL cursorEnd(),deleteChar(),deleteLine(),
backspaceChar(),insertLine();
BOOL saveLine();
void printAll(),printFlags(),fclose(),Insert(),
strncpy(),printNumLines();
void restoreZlinenptr(),SetWindowTitles(),
initFolding(),printYpos();
void DisplayBeep(),saveIfCursorMoved(),CopyMem();
struct Zline *newZline(),*prevLine(),*nextLine();
FILE *fopen();
int fwrite(),getc();
UBYTE *getLastWord();
ULONG fseek();
UWORD recalcTopOfWindow();
/*#ENDFD*/
/*#FOLD:  External Variables  */
/*********************
 *
 * External Variables:
 *
 *********************/

extern struct Editor *actualEditor;
extern struct MsgPort *edUserPort;
/*#ENDFD*/
/*#FOLD:  Globale Variables */
/*********************
 *
 * Globale Variables:
 *
 *********************/
```

```
struct jmpEntry
{
  UBYTE c;
  UBYTE *(*fkt)();
};
/*#ENDFD*/

/***************
 *          *
 * Functions:  *
 *          *
 ***************/

/*#FOLD: saveASCII */
/*********************************
 *
 * saveASCII(name)
 *
 * Save text under name to
 * Diskette .
 *
 * name ^ File name.
 *
 *********************************/

BOOL saveASCII(name)
UBYTE          *name;
{
  register FILE *file;
  register struct Zline *z;

  if   (actualEditor->bufypos)
    if   (!  saveLine(actualEditor->actual))
    {
      DisplayBeep(NULL);
      return (FALSE);
    }

  if   (file = fopen(name,"w"))
  {
    z = actualEditor->zlines.head;
    while (z->succ)
    {
      if (z->len)
        if (fwrite(z + 1,(int)z->len,1,file) != 1)
        {
          /* Error! */
          fclose (file);
          return (FALSE);
        }

      z = z->succ;
    }

    actualEditor->changed = 0;
    printFlags();
    fclose (file);
    return (TRUE);
  }
  else
    return (FALSE);
```

```
}
/*#ENDFD*/
/*#FOLD: loadASCII */
/***********************************
 *
 * loadASCII(name)
 *
 * Load text from disk. The text
 * is inserted after the actual line
 * in the text.
 *
 * name ^ File name.
 *
 ***********************************/

BOOL loadASCII(name)
UBYTE       *name;
{
  UBYTE buf[MAXWIDTH];
  register UBYTE *ptr,*tab;
  int c;
  UWORD rm = actualEditor->rm;
  register FILE *file;
  register UWORD len;
  register struct Zline *z,*zn;

  if (file = fopen(name,"r"))
  {
    /* display name in title line: */
    strncpy(actualEditor->filename,name,
        sizeof(actualEditor->filename));
    SetWindowTitles(actualEditor->window,actualEditor-
>filename,-1L);

    if  (z  = ZLINESPTR(actualEditor->wdy))
    {
      actualEditor->changed = 1;
      printFlags();
    }

    while (!feof(file))
    {
      tab = actualEditor->tabstring;
      ptr = buf;
      len = 0;

      /* read one line: */
      while (!feof(file) && (len < rm))
      {
        if ((c = getc(file)) == EOF)
          break;
        else
          *ptr++ = (UBYTE)c;

        if ((c == TAB) && (actualEditor->tabs))
          do
          {
            tab++;
            len++;
          } while ((*tab) && (len < rm));
        else
        {
```

```
                tab++;
                len++;

                if (c == LF)
                  break;
                else if (c == CR)
                  if (!feof(file) && (len < rm))
                  {
                    if ((*ptr = (UBYTE)getc(file)) == LF)
                      ptr++;
                    else
                      /* set file pointer back: */
                      fseek(file,-1L,1);

                    break;
                  }
              }   /* of if (TAB) */
            }     /* of while */

            if (len = ptr - buf)
            {
              /* Wordwrap? */
              if (!actualEditor->skipblanks)
                if ((len >= rm) && (c != CR) && (c != LF))
                {
                  tab = getLastWord(buf,len);
                  fseek(file,(long)tab - ptr,1);
                  len = (ptr = tab) - buf;
                }

              /* save line now: */
              actualEditor->lineptr = z;  /* Garbage-Collection      */
              if (zn = newZline(len))
              {
                z = actualEditor->lineptr;
                zn->len = len;
                CopyMem(buf,zn  + 1,(ULONG)len);
                actualEditor->num_lines++;
                Insert(&actualEditor->zlines,zn,z);
                z = zn;
                actualEditor->lineptr = NULL;
              }
              else
              {
                actualEditor->lineptr = NULL;
                fclose (file);
                return (FALSE);
              }
            }
          }

          /* calculate fold level: */
          initFolding();

          /* redisplay window: */
          if ((ZLINESPTR(0) == NULL) && (actualEditor->zlines.head-
>succ))
          {
            ZLINESPTR(0) = actualEditor->zlines.head;
            ZLINESNR(0)  = 1;
          }
          restoreZlinenptr();
```

```
    printAll();
    printNumLines();

    fclose (file);
    return (TRUE);
  }
  else
    return (FALSE);
}
/*#ENDFD*/

/************************************************
 *
 * command-Functions:
 *
 * This processes a function group
 *
 * All functions return a pointer to
 * the next command, if no error
 * is encountered, otherwise a negative pointer
 * to the faulty character is returned.
 *
 * Returna a NULL, so this
 * serves as break criteria.
 *
 * A -1 ends the program.
 *
 ************************************************/

/*#FOLD: commandASCII */
UBYTE *commandASCII(str)
register UBYTE     *str;
{
  UBYTE *error = 1L - str,buf[80];
  register UBYTE *name;

  switch (UC(*str))
  {
    case 'L':
      str++;
      SKIPBLANKS(str)
      if (*str == '"')
      {
        /* take file name from command line */
        str++;
        name = buf;
        while ((*name = *str) && (*str != '"')
            && (name < buf + sizeof(buf) - 1))
        {
          name++;
          str++;
        }

        *name = 0;
        if (*str == '"')
          str++;

        name = buf;
      }
      else
        /* use file name form load: */
        name = actualEditor->filename;
```

```
      if (!loadASCII(name))
        return (error);

      break;
    case 'S':
      str++;
      SKIPBLANKS(str)
      if (*str == '"')
      {
        /* get file name from command line */
        str++;
        name = buf;
        while ((*name = *str) && (*str != '"')
            && (name < buf + sizeof(buf) - 1))
        {
          name++;
          str++;
        }

        *name = 0;
        if (*str == '"')
          str++;

        name = buf;
      }
      else
        /* use file name from load: */
        name = actualEditor->filename;

      if (!saveASCII(name))
        return (error);

      break;

    default:
      return (NULL - str);
  }

  return (str);
}
/*#ENDFD*/
/*#FOLD: commandCursor */
UBYTE *commandCursor(str)
register UBYTE      *str;
{
  switch (UC(*str))
  {
    case 'R':
      if (!cursorRight())
        return (NULL);
      break;
    case 'L':
      if (!cursorLeft())
        return (NULL);
      break;
    case 'U':
      if (!cursorUp())
        return (NULL);
      break;
    case 'D':
      if (!cursorDown())
```

```
            return (NULL);
          break;
        case 'E':
          if (!cursorEnd())
            return (NULL);
          break;
        case 'S':
        case 'H':
          if (!cursorHome())
            return (NULL);
          break;
        case 'N':
          if (!cursorDown())
            return (NULL);
          if (!cursorHome())
            return (NULL);
          break;
        case 'P':
          if (!cursorUp())
            return (NULL);
          if (!cursorHome())
            return (NULL);
          break;
        case 'T':
          if (!cursorTop())
            return (NULL);
          break;
        case 'B':
          if (!cursorBottom())
            return (NULL);
          break;

        default:
          return (NULL - str);
      }

      return (str + 1);
}
/*#ENDFD*/
/*#FOLD: commandDelete */
UBYTE *commandDelete(str)
register UBYTE      *str;
{
      switch (UC(*str))
      {
        case 'L':
          if (! deleteLine())
            return (NULL);
          break;
        case 'C':
          if (! deleteChar())
            return (NULL);
          break;
        case 'B':
          if (! backspaceChar())
            return (NULL);
          break;

        default:
          return (NULL - str);
      }
```

```
  return (str + 1);
}
/*#ENDFD*/
/*#FOLD: commandLine */
UBYTE *commandLine(str)
register UBYTE    *str;
{
  switch (UC(*str))
  {
    case 'S':
      if (! insertLine((UBYTE) 0))
        return (1 - str);
      break;

    default:
      return (NULL - str);
  }

  return (str + 1);
}
/*#ENDFD*/
/*#FOLD: commandQuit */
UBYTE *commandQuit(str)
register UBYTE    *str;
{
  return (-1L);
}
/*#ENDFD*/
/*#FOLD: commandSet */
UBYTE *commandSet(str)
register UBYTE    *str;
{
  switch (*str)
  {
    case 't':
      actualEditor->tabs = 0;
      printAll();
      printFlags();
      break;
    case 'T':
      actualEditor->tabs = 1;
      printAll();
      printFlags();
      break;
    case 'i':
    case 'I':
      actualEditor->insert = 1;
      printFlags();
      break;
    case 'o':
    case 'O':
      actualEditor->insert = 0;
      printFlags();
      break;
    case 'b':
      actualEditor->skipblanks= 0;
      printFlags();
      break;
    case 'B':
      actualEditor->skipblanks= 1;
      printFlags();
      break;
```

```
    case 'a':
      actualEditor->autoindent= 0;
      printFlags();
      break;
    case 'A':
      actualEditor->autoindent= 1;
      printFlags();
      break;
    case 'r':
    case 'R':
    {
      register UWORD rm;

      if (*++str == '=')
      {
        str++;
        if  (DIGIT(*str))
        {
          rm = 0;
          while (DIGIT(*str))
          {
            rm = 10*rm + (*str - '0');
            str++;
          }

          if (rm > MAXWIDTH)
            rm = MAXWIDTH;
          else if (rm < 10)
            rm = 10;
        }
        else
          /* Fehler */
          return (NULL - str);
      }
      else
        rm = MAXWIDTH;

      actualEditor->rm = rm;
      --str;
      break;
    }

    default:
      return (NULL - str);
  }

  return (str + 1);
}
/*#ENDFD*/
/*#FOLD: commandExit */
UBYTE *commandExit(str)
register UBYTE    *str;
{
  if (actualEditor->changed)
    if (saveASCII(actualEditor->filename))
      return (-1L);
    else
      return (NULL - str);
  else
    return (-1L);
}
/*#ENDFD*/
```

```
/*#FOLD:  commandBracket  */
UBYTE   *executeCommand();     /* declaration due to recursion!

UBYTE   *commandBracket(str)
register  UBYTE     *str;
{
  register  UBYTE *ptr;

  if  (ptr = executeCommand(str))
  {
    if  (ptr != -1L)
      if  (ptr >= 0x80000000)
        /*  Pointer behind command line '}':  */
        ptr =   1L  - ptr;
      else
        /*  pointer to error: */
        ptr = NULL - ptr;
  }
  else
  {
    /* search for ending '}': */
    ptr =   str;
    while (*ptr &&  (*ptr !=  '}'))
    {
      if  (*ptr ==  '"')
        do
        {
          ptr++;
        } while (*ptr && (*ptr != '"'));

      ptr++;
    }

    if  (*ptr ==  0)
      ptr = NULL;
    else
      ptr++;
  }

  return (ptr);
}
/*#ENDFD*/

/*#FOLD:  messageWaiting  */
/*********************************
 *
 * messageWaiting:
 *
 * Test if there is a message
 * in the UserPort.
 *
 * Returns TRUE if yes.
 *
 *********************************/

BOOL messageWaiting()
{
  register struct IntuiMessage *im;

  im = (struct IntuiMessage *)edUserPort->mp_MsgList.lh_Head;
  while (im->ExecMessage.mn_Node.ln_Succ)
  {
```

```
    if ((im->Class == RAWKEY)
    && !(im->Code & IECODE_UP_PREFIX)) return (TRUE);
    if (im->Class == NEWSIZE) return (TRUE);
    im = (struct IntuiMessage *)im->ExecMessage.mn_Node.ln_Succ;
  }

  return (FALSE);
}
/*#ENDFD*/
/*#FOLD: Functions table */
struct jmpEntry jmpTable[] =
{
  'A',&commandASCII,
  'C',&commandCursor,
  'D',&commandDelete,
  'L',&commandLine,
  'Q',&commandQuit,
  'S',&commandSet,
  'X',&commandExit,
  '{',&commandBracket
};
/*#ENDFD*/
/*#FOLD:  executeCommand  */
/****************************************
 *
 * executeCommand(str)
 *
 * Executes the commands to which str,
 * points. The end is marked by
 * a Null byte.
 *
 * The function returns the following:
 * 0:  no error
 * -1:   Program end
 *  else: pointer to error in string.
 *  or:    negative pointer to '}'
 *
 ****************************************/

UBYTE *executeCommand(str)
register UBYTE      *str;
{
  register UBYTE *ptr,c;
  register UWORD count,n;
  register UBYTE *(*fkt)();

  while (!messageWaiting() && *str)
  {
    SKIPBLANKS(str)

    if  (*str ==  '}')
      return (NULL -  str);

    if (DIGIT(*str))
    {
      count = 0;
      while (DIGIT(*str))
      {
        count = 10*count + (*str - '0');
        str++;
      }
```

```
        if   (count == 0)
          count = 1;
      }
      else
        count = 1;

      /* search function, the command executes: */
      c = UC(*str);
      fkt = NULL;
      for (n = 0; n < sizeof(jmpTable)/sizeof(struct jmpEntry);
n++)
        if (c == jmpTable[n].c)
        {
          fkt = jmpTable[n].fkt;
          break;
        }

      if (fkt)
        while (!messageWaiting() && count--)
        {
          if (ptr = (*fkt)(str + 1))
          {
            if (ptr >= 0x80000000)
              if (ptr == -1L)
                /* Program end */
                return (ptr);
              else
                /* Error encountered: */
                return (NULL - ptr);
            else
              /* If count == 0: Next command, */
              /* else repeat */
              if (count == 0)
                str = ptr;
          }
          else
            /* break */
            return (NULL);
        }
      else
        /* Unknown command */
        return (str);

      saveIfCursorMoved();

      SKIPBLANKS(str)

      if (*str == ';')
        str++;
      else if (*str == '}')
        return (NULL - str);
      else if (*str)
        return (str);
    }

    return (NULL);
}
/*#ENDFD*/
```

A few remarks about the new module:

- The include file `stdio.h` must be added to the modules, since file processing requires the standard C file functions `fopen`, `fclose`, etc.

- The `defines` contain useful macros which convert a character to upper case (`UC`=upper case), to differentiate when a character is handled as a numeral (`DIGIT`), and bypass blank spaces in a string (`SKIPBLANKS`).

- Because of the design of the command set, we don't need an elaborate `switch case` function to differentiate between commands. The first letter of the command and the address of the corresponding function are placed in an array. The data structure of the array is `jmpEntry`. The array is defined after the command functions because the functions must already be defined when the field is initialized (the C compiler must be able to recognize the function addresses).

- The `saveASCII` function expects a pointer to a filename as a parameter, and saves the entire text under this name to disk. The function returns FALSE if an error occurs.

- The same data for `saveASCII` applies to the `loadASCII` function, although it is more complicated because text must be separated into individual lines. The text to be loaded is inserted after the current line of the existing text. If no text exists, the text loads as usual. The variable `z` points to the previous line.

When loading, individual characters are read until the program reaches an end of line character, or when the line is too wide. Because the function `getc` (which reads the characters) works internally with a buffer, the method is no slower than if we loaded an entire text block into a buffer and removed the characters from there. The maximum number of characters per line that can be loaded is a changeable parameter—the new variable `rm` in the `Editor` structure serves this purpose. `rm` displays the right margin.

A set `skipblanks` flag breaks lines if a line is too long to fit on the screen. This break shifts the word causing the problem to the next line. `SkipBlanks` is also known as word wrap. The `getLastWord` function is implemented in the `Edit` module. It returns a pointer to the beginning of the last word of a string.

The editor had an error in it since its first stages of development. The `garbageCollection` function in the `Memory` module reorganizes a memory block. All of the lines can be moved so that this function

checks the `actual` variable and the `zlinesptr` array, and moves a pointer. A pointer to the last line borrows data from the `loadASCII` function, which must also be checked. The `Editor` structure uses this `zlinesptr` array. The `garbageCollection` function checks this pointer. When we have a local variable that points to a line, we save this in `zlineptr` before calling the `newLine` function , which calls the `garbageCollection` function. After the call we access `zlineptr` again, and can be certain that this pointer points to the same line. The error occurs in the `saveLine` function, when the `newLine` function is called while a local variable contains a pointer to a line. We will correct this error.

After all of the lines load, `initFolding` recalculates the fold levels of all of the lines, and the window contents are redisplayed.

- The functions that execute individual commands are called `commandXYZ`, where `XYZ` represents the command group. A pointer to the second letter of the command is given in this function. A pointer is returned which usually points to the location following the command. However, special cases can arise: When the function returns zero, this serves as the break criteria for the command interpreter. A value of -1 exits the editor (i.e., X and Q). When the program otherwise returns a negative value, an error occurs. You get a pointer to the error when you multiply the return value by -1 or when you pass it zero.

- The currently implemented commands consist of corresponding function calls, which include other modules. The ASCII commands are somewhat more extensive because a character string must be handled. The `sr=nn` function in the `Set` commands sets the flags. You can set the right margin for word wrap from this function. The value can be a value from 10 to `MAXWIDTH`.

- Bracketed commands call the `commandBracket` function, which recursively calls the `executeCommand` function. The `executeCommand` function returns if an error occurs, or if brackets appear in the command line. In the latter case, execution continues following the closing bracket (}). Problems can occur if the user stops the bracketed command during execution. In this case command execution should continue after the closing bracket (}). The `commandBracket` function searches for a closing bracket (}) and returns a pointer to the following characters. The user can stop the command without problem by pressing a key, because this break criteria stays the same.

- The `messageWaiting` function checks for a user-invoked command break. This function executes either if a key is pressed (<Shift>, <Ctrl> or <Alt>, because these only send `RAWKEY`), or if the window size changes. In the latter case, the output becomes incorrect, because some of the values calculated for output in the `initWindowSize` function probably changed.

- The `executeCommand` function controls the execution of each command. It expects a pointer to the command sequence as a parameter. It returns zero if no error occurs (similar to the command functions), -1 on program end, or a pointer to the error. If the returned value is negative, it treats the negative number as a pointer to a closing bracket (}). When the `commandBracket` function calls the `executeCommand` function, it handles the number as the end of a bracketed command sequence. When the function is called from the main program, it handles the number as an error because no opening bracket ({) occurred in the command sequence.

Because pointers in C are handled as values without prefixes, we can't test any pointers less than zero. Instead we must test the pointer for a value greater than 0x800000000 (currently the smallest negative number available).

Fixing the bug Next we will correct the error mentioned above, which lies in the `Memory.c` module. Insert the following lines in the `garbageCollection` function, so that the `actualEditor->actual` pointer makes the proper test:

```
if   (actualEditor->lineptr == z)
                actualEditor->lineptr = fz;
```

Now add the following includes to the `Editor.h` file:

```
#include <stdio.h>
#include <intuition/intuitionbase.h>
```

The positions and lengths of each value of the info line must be stated as `defines`:

```
#define INFO_XPOS_INC 2
#define INFO_XPOS_LEN 4
#define INFO_YPOS_INC 9
#define INFO_YPOS_LEN 4
#define INFO_NUMOFLINES_INC 16
#define INFO_NUMOFLINES_LEN 4
#define INFO_MINFOLD_INC 22
#define INFO_MINFOLD_LEN 2
#define INFO_MAXFOLD_INC 25
#define INFO_MAXFOLD_LEN 2
#define INFO_FLAGS_INC 29
#define INFO_FLAGS_LEN 5
```

The _INC defines state the number of characters in the value removed from the beginning of the info line, and the _LEN defines state the number of characters in the value. The following elements extend the Editor structure:

```
  UBYTE gc_SIBuffer[256];
  struct StringInfo gc_GadgetSI;
  struct Gadget G_Command;
  UBYTE gi_Text[36];
  struct IntuiText gi_IText;
  struct Gadget G_Info;
  ULONG ip_xpos;
  ULONG ip_ypos;
  ULONG ip_numzlines;
  ULONG ip_minfold;
  ULONG ip_maxfold;
  ULONG ip_flags;
  ULONG ip_topedge;
  UBYTE filename[80];
  UWORD rm;
  struct Zline *lineptr;
};
```

The first six elements contain the gadget structures for the command and info lines. Next follow the positions of the individual values of the info line, the current filename initialized from the loadASCII function, the right margin and the help pointer to a line viewed by the garbageCollection function.

We now turn our attention to the main module Editor.c, which requires the addition of another include file:

```
#include <intuition/intuitionbase.h>
```

Two new defines follow, they state the argument template returned when the user enters Editor ? from the CLI:

```
#define UT1 "USAGE: Editor [Flags] <Filename>"
#define UT2 "Flags: [-a][-A][-b][-B][- i|I][- o|O][- r|R[=n]][-
t][-T]"
```

Since the editor now has built in text loading capability, you can start the editor from the CLI by entering the editor name, a space and the filename:

```
editor myfile
```

You can also enable and disable flags from the CLI as the editor loads. For example, instead of loading the editor, pressing <Esc> and entering SA to enable auto indent, the following loads the editor and sets the auto indent flag:

```
editor -a
```

Let's look at the other external functions:

```
void strncpy(),CloseFont(),puts();
void printXpos(),printYpos(),printInfo(),DisplayBeep();
struct TextFont *OpenFont();
BOOL handleKeys(),ActivateGadget(),loadASCII();
UBYTE *executeCommand(),*commandSet();
```

Remember that the `handleKeys` function is now of type `BOOL`. You remember the reason for this when we changed the `Cursor.c` module. We defined two gadgets with global variables. These gadgets control the command and info lines. The `G_command` string gadget represents the command line, while the `G_Info` boolean gadget represents the info line:

```
UBYTE gc_SIBuffer[256];
UBYTE gc_UndoBuffer[256];
struct StringInfo gc_GadgetSI =
{
  gc_SIBuffer,
  gc_UndoBuffer,
  0,
  256,
  0,
  0,0,0,0,0,
  0,
  NULL,
  NULL
};

SHORT gc_BorderVectors[4] = {0,0,350,0};
struct Border gc_Border =
{
  -1,-1,
  1,0,JAM1,
  2,
  gc_BorderVectors,
  NULL
};

struct Gadget G_Command =
{
  NULL,
  2,-9,
  -302,9,
  GADGHCOMP | GRELBOTTOM | GRELWIDTH,
  RELVERIFY | BOTTOMBORDER,
  STRGADGET,
  (APTR)&gc_Border,
  NULL,
  NULL,
  0,
  (APTR)&gc_GadgetSI,
  2,
  NULL
};

SHORT gi_BorderVectors[14] = {-1,0,-
1,10,0,10,0,0,298,0,298,1,283,1};
```

```
struct Border gi_Border =
{
  0,0,
  1,0,JAM1,
  7,
  gi_BorderVectors,
  NULL
};

struct TextAttr TOPAZ80 =
{
  (STRPTR)"topaz.font",
  TOPAZ_EIGHTY,0,0
};

UBYTE gi_Text[36] = "X=   1 Y=   1 #=   0 [ 0, 0] IcATB";
struct IntuiText gi_IText =
{
  1,0,JAM2,
  4,2,
  &TOPAZ80,
  gi_Text,
  NULL
};

struct Gadget G_Info =
{
  &G_Command,
  -298,-10,
  280,10,
  GADGHNONE | GRELBOTTOM | GRELRIGHT,
  RELVERIFY | BOTTOMBORDER,
  BOOLGADGET,
  (APTR)&gi_Border,
  NULL,
  &gi_IText,
  0,
  NULL,
  1,
  NULL
};

struct NewWindow newEdWindow =
{
  0,0,640,200,
  AUTOFRONTPEN,AUTOBACKPEN,
  REFRESHWINDOW | MOUSEBUTTONS | RAWKEY | CLOSEWINDOW | NEWSIZE |
GADGETUP,
  WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH | WINDOWCLOSE |
SIZEBBOTTOM
  | SIMPLE_REFRESH | ACTIVATE,
  NULL,NULL,
  NULL,
  NULL,NULL,
  320,50,640,200,
  WBENCHSCREEN
};

struct TextFont *infoFont = NULL;
```

The window allows GADGETUP events, so the program receives a report when the user presses the <Return> key in the string gadget, and thus executes a command sequence. Since we want to be able to open more editor windows later, we must copy sections of this gadget to the Editor structure, because no two windows can use the same gadgets. We don't need to copy all of the structures: A border structure; the command line buffer (SIBuffer) is needed so we can enter two different command sequences; the StringInfo structure; the info line text; the Intuitext structure, which points to this; and the two gadget structures, for two windows.

The gadgets and other structures are copied into the OpenEditor function in the Editor structure. This happens directly after the memory allocation for the Editor structure, because the gadgets must be joined to the NewWindow structure so that they also appear in the window. In addition, remember to set the structure pointers to one another, because this no longer makes sense after copying them into the Editor structure. The following OpenEditor function avoids initializing the rest of the parameters since they are the same:

```
struct Editor *OpenEditor()
{
  register struct Editor *ed = NULL;
  register struct Window *wd;
  register struct RastPort *rp;
  ULONG flags;

  /* IDCMPFlags saved, sets it to zero in structure
    => use your own UserPort! */
  flags = newEdWindow.IDCMPFlags;
  newEdWindow.IDCMPFlags = NULL;

  /* Memory for editor structure: */
  if (ed = malloc(sizeof(struct Editor)))
  {
    /* Gadgets initialization: */
    ed->gc_SIBuffer[0]  = 0;
    ed->gc_GadgetSI     = gc_GadgetSI;
    ed->G_Command       = G_Command;
    ed->gi_IText        = gi_IText;
    ed->G_Info          = G_Info;
    strncpy(ed->gi_Text,gi_Text,sizeof(ed->gi_Text));

    /* set pointer: */
    ed->gc_GadgetSI.Buffer     = ed->gc_SIBuffer;
    ed->G_Command.SpecialInfo  = (APTR) &(ed->gc_GadgetSI);
    ed->gi_IText.IText         = ed->gi_Text;
    ed->G_Info.NextGadget      = &(ed->G_Command);
    ed->G_Info.GadgetText      = &(ed->gi_IText);
    newEdWindow.FirstGadget    = &(ed->G_Info);
    newEdWindow.Title          = ed->filename;
    ed->filename[0]          = 0;

    /* Window open: */
    if (wd = OpenWindow(&newEdWindow))
    {
```

```
ed->window = wd;
ed->rp = (rp = wd->RPort);

/* Write mode set: */
SetDrMd(rp,JAM2);
SetAPen(rp,FGPEN);
SetBPen(rp,BGPEN);

/* UserPort established: */
wd->UserPort = edUserPort;
ModifyIDCMP(wd,flags);

/* Parameter initialization: */
NewList(&(ed->block));
NewList(&(ed->zlines));

ed->num_lines  = 0;
ed->actual     = NULL;
ed->buflen     = 0;
ed->bufypos    = 0;
ed->buflastchar= ' ';
ed->leftpos    = 0;
ed->xpos       = 1;
ed->ypos       = 1;
ed->wdy      = 0;
ed->changed    = 0;
ed->insert     = 1;
ed->tabs       = 1;
ed->skipblanks  = 1;
ed->autoindent  = 1;
ed->minfold    = 0;
ed->maxfold    = 0;
ed->rm       = MAXWIDTH;

{
  register UBYTE *ptr;
  register struct Zline **zptr;
  register UWORD n,*pnr;

  /* zlinesptr/nr-Array initialization: */
  for (n = 0, zptr = ed->zlinesptr, pnr = ed->zlinesnr;
      n < MAXHEIGHT; n++, zptr++, pnr++)
  {
    *zptr = NULL;
    *pnr  = 1;
  }

  /* Tab initialization: */
  ptr = ed->tabstring;
  *ptr++ = 1;
  for (n = 1; n < MAXWIDTH; n++)
    if (n % 3)
      *ptr++ = 1;
    else
      *ptr++ = 0;
}

{
  register struct Editor *oldae;

  ed->wch = 0;
  initWindowSize(ed);
```

```
            oldae        = actualEditor;
            actualEditor = ed;
            printInfo();
            actualEditor = oldae;
          }
        }
        else
        {
          free(ed);
          ed = NULL;
        }
      }

      newEdWindow.IDCMPFlags = flags;
      return (ed);
}
```

The info line is displayed at the end of `OpenEditor` so that this agrees with the additional values. The `actualEditor` variable turns towards the re-opened editor because the output functions assume that `actualEditor` points to the info line of the current editor.

Our main program requires some major additions. The `main` function expects the `argc` and `argv` arguments, which encompass all of the arguments given to the program from CLI access. At the beginning of the program, before the libraries open, the program tests to see whether the editor was called with `Editor ?`. If so, the program displays the argument template and returns to the CLI. The `puts` function displays a string without the format specifications and escape sequences of the `printf` function.

After the libraries are open the character set for the info line is accessed, and a pointer to it is placed in the `infoFont` variable. Before the editor is opened, the program determines the maximum height of the screen and enters this in the `NewWindow` structure, giving the window the maximum size when it is opened. The `ActiveScreen` variable of the `IntuitionBase` structure acts as a pointer to the active screen.

After the editor window opens, the program reads the arguments entered from the CLI (if any) and executes these arguments through the command line. The order of the arguments must follow the template. Otherwise the program ends and an error message appears. The one error that loads the editor anyway occurs when the editor tries to find the filename stated from the CLI and fails.

The main loop stays the same as before. However, the `handleKeys` function returns a value of type BOOL. If this value is FALSE, the program stops. `MOUSEBUTTON` and `NEWSIZE` messages display the cursor position with `printXpos` and `printYpos`, if these are changed.

The GADGETUP message check is new. This began execution of the command line. If an error occurs, the cursor of the gadget moves to the location of the error and the string gadget becomes active. Because the GADGETUP check does not know which gadget caused this message, you can access the command line by clicking on the command line with the mouse. This allows the user to execute a command sequence that has already been entered.

The end of the program has also changed. All of the variables that were tested are set to zero. The reason for this can be found in a utility program named SHELL. This program is an improvement to the CLI that allows the definition of commands as resident (the commands are retained in memory, regardless of how often they are executed). The advantage here is that the command does not have to be loaded first. In addition, the resident command requires less memory than if you copied the command to the RAM disk as a disk file. If the command was copied to the RAM disk, it would have to be loaded before it could execute. If it is defined as resident, it appears in memory once, and only becomes active when the user calls it.

Commands or programs to be defined as resident must fulfill certain criteria. Never assume that variables are initialized because their contents can be derived from an earlier call. To get around this potential problem, all important variable contents return to their starting values when the program ends. This ensures that the editor finds initialized starting values every time the user executes the editor. Now on to the new main function:

```
main(argc,argv)
int  argc;
UBYTE     *argv[];
{
  register struct Editor *ed;
  BOOL running = TRUE;
  register struct IntuiMessage *imsg;
  ULONG signal,class,mouseX,mouseY;
  UWORD code,qualifier;
  APTR iaddress;

  /*  Was Editor started with "Editor ?" , the Text output */
  if   (argc ==  2)
    if   (*argv[1] == '?')
    {
      puts(UT1);
      puts(UT2);
      goto Ende;
    }

  /* Libraries open: */
  if ( !(IntuitionBase = OpenLibrary("intuition.library",REV)))
    goto Ende;
  if ( !(GfxBase = OpenLibrary("graphics.library",REV)))
    goto Ende;
```

```
  if ( !(DosBase = OpenLibrary("dos.library",REV)))
    goto Ende;
  if ( !(OpenDevice("console.device",-1L,&ioStdReq,0L)))
    ConsoleDevice = ioStdReq.io_Device;
  else
    goto Ende;

  /* Font for Infoline open: */
  if (! (infoFont = OpenFont(&TOPAZ80))) goto Ende;

  /* UserPort open */
  if (! (edUserPort = CreatePort(NULL,0L))) goto Ende;

  /*  Max height for window calculated: */
  newEdWindow.Height =  newEdWindow.MaxHeight =
        ((struct  IntuitionBase *)IntuitionBase)->ActiveScreen-
>Height;

  /* Editor list initialization: */
  NewList(&editorList);

  /* Open first editor window: */
  if (ed = OpenEditor())
  {
    AddTail(&editorList,ed);
    actualEditor = ed;
  }
  else
    goto Ende;

  /* Command line reading: */
  if (argc >= 2)
  {
    register UWORD n = 1;

    while (n  < argc)
    {
      if  (*argv[n] == '-')
      {
        if  (commandSet(argv[n] + 1) >= 0x80000000)
        {
          DisplayBeep(NULL);
          break;
        }
      }
      else
      {
        if  (loadASCII(argv[n]))
          printAll();
        else
          DisplayBeep(NULL);

        /*  File name end input! */
        n++;
        break;
      }

      n++;
    }

    if (n < argc)
    {
```

```
    /*  Error! */
    puts(UT1);
    puts(UT2);
    goto Ende;
  }
}

/*#FOLD:  Main loop */
do
{
  /* Set cursor: */
  Cursor();

  signal = Wait(1L << edUserPort->mp_SigBit);

  /* erase cursor again: */
  Cursor();

  while (imsg = GetMsg(edUserPort))
  {
    class     = imsg->Class;
    code      = imsg->Code;
    qualifier = imsg->Qualifier;
    iaddress  = imsg->IAddress;
    mouseX    = imsg->MouseX;
    mouseY    = imsg->MouseY;

    ReplyMsg(imsg);

    /* Event processing: */
    switch (class)
    {
      case RAWKEY:
      {
        register struct IntuiMessage *im1,*im2;

        if (! (code & IECODE_UP_PREFIX))
        {
          inputEvent.ie_Code     = code;
          inputEvent.ie_Qualifier = qualifier;
          if ((inputLen = RawKeyConvert(
                &inputEvent,inputBuffer,MAXINPUTLEN,NULL)
                ) >= 0)
            running = handleKeys(inputBuffer,inputLen);
        }

        /* run-on suppressed: */
        im1 = (struct IntuiMessage *)
            edUserPort->mp_MsgList.lh_Head;
        while (im2 = (struct IntuiMessage *)
                 im1->ExecMessage.mn_Node.ln_Succ)
        {
          if (im2->ExecMessage.mn_Node.ln_Succ == NULL) break;
          if (im1->Class != RAWKEY) break;
          if (!(im1->Qualifier & IEQUALIFIER_REPEAT)) break;
          if (im2->Class != RAWKEY) break;
          if (!(im2->Qualifier & IEQUALIFIER_REPEAT)) break;

          /* Message reply: */
          im1 = GetMsg(edUserPort);
          ReplyMsg(im1);
```

```
    im1 = (struct IntuiMessage *)
        edUserPort->mp_MsgList.lh_Head;
  }

  break;
}
case MOUSEBUTTONS:
  {
    register WORD x,y;

    if (mouseX <= actualEditor->xoff)
      x = 0;
    else
      x = (mouseX - actualEditor->xoff)
        / actualEditor->cw;
    if (++x >= actualEditor->wcw)
      x = actualEditor->wcw - 1;
    x += actualEditor->leftpos;
    if (x > MAXWIDTH)
      x = MAXWIDTH;

    if (mouseY <= actualEditor->yoff)
      y = 0;
    else
      y = (mouseY - actualEditor->yoff)
        / actualEditor->ch;

    if ((y < actualEditor->wch)
      && (actualEditor->zlinesptr[y]))
    {
      actualEditor->xpos = x;
      actualEditor->wdy  = y;
      actualEditor->ypos = actualEditor->zlinesnr[y];

      printXpos();
      printYpos();
    }
  }

case NEWSIZE:
  initWindowSize(actualEditor);

  /* check if cursor still in window! */
  if (actualEditor->xpos > actualEditor->leftpos
                    +actualEditor->wcw)
  {
    actualEditor->xpos = actualEditor->leftpos
                    + actualEditor->wcw;
    printXpos();
  }

  if (actualEditor->wdy >= actualEditor->wch)
  {
    actualEditor->ypos = actualEditor->zlinesnr
      [(actualEditor->wdy  = actualEditor->wch - 1)];
    printYpos();
  }

  break;

case REFRESHWINDOW:
  BeginRefresh(actualEditor->window);
```

```
                printAll();
                EndRefresh(actualEditor->window,TRUE);
                break;

              case GADGETUP:
              {
                register UBYTE *ptr;
                register SHORT pos,hw;

                if (ptr = executeCommand(actualEditor->gc_SIBuffer))
                  if (ptr == -1L)
                    running = FALSE;
                  else
                  {
                    if   (ptr >= 0x80000000)
                      ptr =   NULL - ptr;

                    /* set cursor in StringGadget to error */
                    pos = ptr - actualEditor->gc_SIBuffer;
                    actualEditor->gc_GadgetSI.BufferPos = pos;
                    if (pos < actualEditor->gc_GadgetSI.DispCount)
                      actualEditor->gc_GadgetSI.DispPos = 0;
                    else
                      actualEditor->gc_GadgetSI.DispPos = pos
                          - actualEditor->gc_GadgetSI.DispCount / 2;

                    /* Gagdet aktivated: */
                    ActivateGadget(&(actualEditor->G_Command),
                                actualEditor->window,NULL);
                    DisplayBeep(NULL);
                  }

                break;
              }

              case CLOSEWINDOW:
                running = FALSE;
                break;

              default:
                printf("Not a processable event: %lx\n",class);

            } /* of case */

            saveIfCursorMoved();
          } /* of while (GetMsg()) */
        } while (running);
        /*#ENDFD*/

      Ende:
        /* Close all editor windows: */
        while (ed = RemHead(&editorList))
          CloseEditor(ed);

        /* Close UserPort: */
        if (edUserPort)
        {
          DeletePort(edUserPort);
          edUserPort = NULL;
        }

        /* Font closed: */
```

```
  if (infoFont)
  {
    CloseFont(infoFont);
    infoFont = NULL;
  }

  /* Close Libraries: */
  if (DosBase)
  {
    CloseLibrary(DosBase);
    DosBase = NULL;
  }
  if (GfxBase)
  {
    CloseLibrary(GfxBase);
    GfxBase = NULL;
  }
  if (IntuitionBase)
  {
    CloseLibrary(IntuitionBase);
    IntuitionBase = NULL;
  }
}
```

CLI arguments in the editor

The `commandSet` function reads the CLI command so that you can set flags from the CLI instead of in the editor's command line (we mentioned this earlier). When entering the set flags from the CLI, remember to separate each argument with a space, and enter a dash instead of an S. The following CLI command loads the editor, sets the right margin at 60, disables both `SkipBlanks` and tabs, and loads the ASCII file named `Floatingtext.ASC` into the editor:

```
Editor -r=60 -b -t Floatingtext.ASC
```

There are also a few changes in the `Cursor.c` module. First let's examine some of the new `Defines`:

```
#define CEND '@'
#define ESC 27
#define TAB 9
#define REPCOM 7
```

Because flag switching now occurs through the command line, we can remove the control code flag switching. Delete the `Defines` `TABMODE`, `WRITEMODE`, and `AUTOINDENT`.

The following external functions are also new. We need these commands for the info and command lines:

```
void printXpos(),printYpos(),printFold(),printFlags();
BOOL ActivateGadget();
UBYTE *executeCommand();
```

The following functions affect the cursor position or other info line values (`cursorLeft`, `cursorRight`, `cursorUp`, `cursorDown`,

halfPageUp, halfPageDown, cursorHome, enterFold, exitFold) so they must call the corresponding functions (printXpos, printYpos, printFold). Three new functions control cursor movement, which place the cursor at the end of the line, the beginning of the text or at the end of the text:

```
/*#FOLD: cursorEnd */
/*************************************
 *
 * cursorEnd:
 *
 * Set the cursor on the last line.
 *
 *************************************/

BOOL cursorEnd()
{
  register UBYTE *ptr,*tab;
  register UWORD len,pos;
  register struct Zline *z;

  if (actualEditor->bufypos)
    pos = actualEditor->buflen + 1;
  else
    if (z = ZLINESPTR(actualEditor->wdy))
    {
      /* remove CR from lenght: */
      len = z->len;
      ptr = ((UBYTE *)(z + 1)) + len - 1;
      if (len)
        if (*ptr == CR)
          len--;
        else if (*ptr == LF)
          if (--len)
            if (*--ptr == CR)
              len--;

      if (actualEditor->tabs)
      {
        ptr = ((UBYTE *)(z + 1));
        tab = actualEditor->tabstring;
        pos = 1;

        while ((len--) && (pos < MAXWIDTH))
          if (*ptr++ == TAB)
            do
            {
              tab++;
              pos++;
            } while ((*tab) && (pos < MAXWIDTH));
          else
          {
            tab++;
            pos++;
          }
      }
      else
        pos = len + 1;
    }
    else
```

```
      return (FALSE);

  actualEditor->xpos = pos;
  if (pos >= (actualEditor->leftpos + actualEditor->wcw))
    scrollLeft(pos + 1 - (actualEditor->leftpos
                + actualEditor->wcw));
  else if (pos <= actualEditor->leftpos)
    scrollRight(actualEditor->leftpos + 1 - pos);

  printXpos();

  return (TRUE);
}
/*#ENDFD*/
```

`cursorEnd` is the most complicated function because the end of the line must be calculated from the tabs. The `cursorTop` function counts the lines that are in the top half of the window and scrolls the window down by this value. The `cursorBottom` function works in the same manner except that the program must be prepared for the possiblity of no text in the bottom of the window. Insert these three functions following the `cursorHome` function.

The `handleKeys` function now returns a value of type BOOL:

```
BOOL handleKeys(buf,len)
```

In the second `switch case` function, at the location where the CSI sequence check occurs, insert three lines following the check for CURSORHOME:

```
case CEND:
   cursorEnd();
   break;
```

In the second `switch case` function, at the location where control code checking occurs, insert the following lines before the check for CFOLD:

```
case ESC:
          /* Rest of buf transferred: */
          {
            register UBYTE *sibuf;
            register UWORD cnt;

            sibuf = actualEditor->gc_SIBuffer;
            while (len)
            {
              *sibuf++ = *buf++;
              len--;
            }
            *sibuf = 0;

            cnt = (sibuf - actualEditor->gc_SIBuffer);
            actualEditor->gc_GadgetSI.BufferPos      = cnt;
            actualEditor->gc_GadgetSI.NumChars       = cnt;
```

```
      actualEditor->gc_GadgetSI.DispPos  = 0;
    }

    /* Gagdet aktivated: */
    if (!ActivateGadget(&(actualEditor->G_Command),
                  actualEditor->window,NULL))
      DisplayBeep(NULL);
    break;
  case REPCOM:
  {
    register UBYTE *ptr;
    register SHORT pos,hw;

    if (ptr = executeCommand(actualEditor->gc_SIBuffer))
      if (ptr == -1L)
        return (FALSE);
      else
      {
        if   (ptr >= 0x80000000)
          ptr = NULL - ptr;

        /* Cursor in StringGadget in error set */
        pos = ptr - actualEditor->gc_SIBuffer;
        actualEditor->gc_GadgetSI.BufferPos = pos;
        if (pos < actualEditor->gc_GadgetSI.DispCount)
          actualEditor->gc_GadgetSI.DispPos = 0;
        else
          actualEditor->gc_GadgetSI.DispPos = pos
              - actualEditor->gc_GadgetSI.DispCount / 2;

        /* Gagdet aktivated: */
        ActivateGadget(&(actualEditor->G_Command),
                    actualEditor->window,NULL);
        DisplayBeep(NULL);

        /* return now: */
        return (TRUE);
      }

    break;
  }
```

When `Esc` is pressed the rest of the characters entered from the keyboard are written to the active command line. This character transfer makes the most sense since you can configure the keyboard. A key could be configured so that when pressed, it sends <Esc> then a command sequence. All the user has to do is press <Return> to enter the sequence.

The second `switch case` function handles the case if the user pressed <Ctrl><G> (`REPCOM`), which repeatedly executes the command currently in the command line. In the case of an error, the cursor of the command line stops at the error and activates the window, just like the `GADGETUP` check in the main program.

You must erase the `TABMODE`, `WRITEMODE` and `AUTOINDENT` checks, because these flags can now be set through the command line.

This keeps the user from pressing the wrong key, and allows the user to insert additional control codes directly into the text. A special feature of our editor is that we can read text which consists of any characters.

When examining the editor we determined that the <Return> key sends a CR. However, other CLI commands (e.g., `TYPE`) cannot make use of lines ending with just a CR. We always add a linefeed (LF) as the end of line so that text written from the editor can be used in other programs. The CR/LF check looks like the following code:

```
case LF:
case CR:
   if (! insertLine((UBYTE) LF))
      DisplayBeep(NULL);
   break;
```

This concludes the `Cursor.c` module. Now we can turn to the final changes needed in the `Edit.c` module. Start by adding a few new external functions:

```
void printXpos(),printYpos(),printFold();
void printNumLines(),printFlags();
BOOL cursorEnd();
```

This module has the calls to the functions corresponding to changes that occur in the info line (`printXpos`, `printYpos`, `printFold`, `printNumLines` and `printFlags`).

A local pointer to a line in `lineptr` of the `Editor` structure must be saved by the `saveLine` function before the `newLine` function is called. The corresponding program segment is as follows:

```
  /* use new line: */
  if (EVENLEN(len) != EVENLEN(line->len))
  {
    actualEditor->lineptr = line;
/* Garbage-Collection */
    if (line = newZline(len))
    {
      old = actualEditor->lineptr;
      Insert(&actualEditor->zlines,line,old);
      line->flags = old->flags & ~ZLF_USED;
      deleteZline(old);

      /* now possible to convert pointer: */
      for (l = 0, zptr = actualEditor->zlinesptr;
          l <= actualEditor->wch; l++, zptr++)
        if (*zptr == old)
        {
          *zptr = line;
          break;
        }

      actualEditor->lineptr = NULL;
    }
    else
```

```
    {
      actualEditor->lineptr = NULL;
      return (FALSE);
    }
  }
  else
  {
    line->len    = len;
    line->flags &= ~ZLF_USED;
  }
```

Remember to place the `printYpos` and `printFlags` calls in this function, because `changed` is set. A call for `printNumLines` also belongs in the `getBufferPointer` function because a first line is redisplayed. The `deleteChar` and `backspaceChar` functions change so that TRUE is always returned. The difference between TRUE and FALSE is unfortunately not confirmed here. This often occurs owing to an interruption during the execution of a command sequence.

A small change in the `insertLine` function must be made, otherwise the `LS` (line split) command will not execute correctly:

```
    /* deconvert line: */
    p2 = buf + (len = deconvertLine(buf,actualEditor-
>buffer,inc));
    if (*p2 = c)
      len++;
```

This lets the editor divide lines without inserting characters. The end of the function redisplays the info line's values:

```
printXpos();
printYpos();
printNumLines();
printFlags();
```

If `minfold` and `maxfold` have changed, `printFold` must be called at the corresponding place.

A small error exists in the `deleteZline` function that we first noticed when implementing the info line. Remember that we had miscalculated the Y position, and since no info line existed at that time, the debugger was the only way to find the error. Something similar has happened in the `deleteZline` function, and at the location determined by the pointer to the next and previous lines:

```
prevnr = ZLINESNR(actualEditor->wdy);
prev   = prevLine(line,&prevnr);
nextnr = ZEILENNR(actualEditor->wdy);
next   = nextLine(line,&nextnr);
```

We assumed that the next line had the same number as the current line, because the current line was erased. The above command sequence must be changed as follows:

```
    prevnr= ZLINESNR(actualEditor->wdy);
    prev  = prevLine(line,&prevnr);
    next  = nextLine(line,&nextnr);
    nextnr= ZLINESNR(actualEditor->wdy);
```

The fact that `nextnr` doesn't initialize when `nextLine` executes causes no problem, because the value from `nextnr` only increments to one in the `nextLine` function.

The `getLastWord` function is new. This function returns the pointer to the last word from a given string. Insert this function before the `insertChar` function:

```
/*#FOLD:  getLastWord */
/*************************************
 * getLastWord(str,len)
 * Returns pointer to last
 * word in a string.
 * str ^ String.
 * len = buffer length.
 *************************************/

UBYTE *getLastWord(str,len)
register UBYTE    *str;
register UWORD        len;
{
  register UBYTE c,*lwb,*lwn,*end;

  lwb = NULL;          /* Last Word is space  */
  lwn = NULL;          /* Last Word normal */
  end = str + len - 1;     /* pointer to string end */
  for (str += len; len; len--)
  {
    if ((c = *--str) == ' ')
    {
      lwb = str;
      break;
    }
    else if (! ((c >= 'A') && (c <= 'Z')
          || (c >= 'a') && (c <= 'z')
          || (c >= '0') && (c <= '9')
          || (c >= 192) || (c == '_')))
      if (lwn == NULL)
        lwn = str;
  }

  if (lwb == NULL)
    if (lwn)
      lwb = lwn;
    else
      lwb = end;
  else
    if (lwn)
      if ((end - lwb) > MAXWIDTH / 5)
        if ((end - lwn + 5) < (end - lwb))
          lwb = lwn;

  return (lwb + 1);
}
/*#ENDFD*/
```

The function searches for the last character preceded by a blank (`lwb` = last word blank), as well as the last character not preceded by a letter, a number, an underline or an international character (`lwn` = last word normal). The `lwb` value is the default. If `lwb` is too far from the end of string (1/5 of the maximum width = 16 characters), and if a proper character is found for `lwn`, the function gives `lwn` first priority provided it is at least five characters closer to the end of string than `lwb`. Words divided with a hyphen are split at the hyphen, as long as the first section of the word is longer than five characters. A negative number like "-123" is ignored, since no characters exist to the left of the minus sign.

The `insertChar` function needs some fine-tuning. When you insert tabs at the beginning of a line and the line reaches the maximum width, no more tabs can be inserted. However, the cursor moves anyway. To avoid this, the `xadd` variable can be reset if the line length goes beyond the maximum width. Expand the line `xadd -= ...` as follows:

```
/* num = Number of spaces to insert: */
if (actualEditor->buflen + num > MAXWIDTH)
{
  xadd -= num + actualEditor->buflen - MAXWIDTH;
  num = MAXWIDTH - actualEditor->buflen;
}
```

`WordWrap` executes before inserting a character in the line:

```
/* besides a normal character: */
if ((! actualEditor->skipblanks)
  &&  (actualEditor->xpos >= actualEditor->rm))
{
  /* Word-Wrap when SkipBlanks == 0 */
  actualEditor->xpos = getLastWord(actualEditor->buffer,
                                   actualEditor->buflen)
                     - actualEditor->buffer + 1;
  if (! insertLine((UBYTE) 0))
    return (FALSE);

  if (! saveLine(actualEditor->actual))
    DisplayBeep(NULL);

  cursorEnd();

  if (! getBufferPointer(&ptr,&inc,&rest))
    return (FALSE);
}

if (actualEditor->insert)
...
```

The original function returns after the character is inserted in the line. The `initFolding` function should be inserted at the end of the `Edit` module:

```
/*#FOLD:  initFolding */
/*********************************
 * initFolding:
 * calculate the fold level of all
 * lines from text start.
 *********************************/

void initFolding()
{
  register struct Zline *z;
  register UWORD level,l;

  z = actualEditor->zlines.head;
  level = 0;
  while (z->succ)
  {
    z->flags = (z->flags & ~ZLF_FOLD) | (level & ZLF_FOLD);
    if (l = getFoldInc(z))
    {
      z->flags |= ZLF_FSE;
      level += l;
    }

    z = z->succ;
  }
}
/*#ENDFD*/
```

This completes the final changes. Remove the `test.c` module from the `makefile`. Replace it with the `Command.c` module and compile it. You can find source text and the finished program on the optional disk for this book, in the `V0.6` directory. Appendix A contains the complete source text for all of the modules..

Now you can load any text; don't use the `TestText` file. Access the editor from the CLI and load the `Command.c` file. If loading from the optional disk for this book, you'd enter the following CLI commands:

```
cd V0.6
Editor src/Command.c
```

We added folds to this and all other modules for maximum readability. You now have an overview of all of the functions in this module. Pressing <Ctrl><F> moves you deeper into a fold, and pressing <Ctrl><E> moves you toward the top fold level.

From this point, you're on your own. You now have all the elements for programming an editor. In addition, you have a finished product for editing your own source texts. We mentioned a few extra commands that could be added later, and listed some of the elements of a programmable editor. With this knowledge and some extra study, you could add extended features to your own editor, such as menus, block functions and more.

4.4 Editor Command Set

The following overview lists the commands available in the final version of the editor listed in this book.

4.4.1 Editor Keyboard Commands

Cursor movement		
	<Cursor up/down/left/right>	moves cursor one line/character in that direction
	<Shift><Cursor right/left>	cursor to beginning/end of current line
	<Shift><Cursor up/down>	cursor one-half page up/down
	<Delete>	deletes character under cursor
	<Backspace>	deletes character to left of cursor
	<Tab>	inserts tab
Control keys	<Ctrl><B>	deletes current line
	<Ctrl><E>	exits current fold
	<Ctrl><F>	enters fold
	<Ctrl><G>	re-executes current command
	<Ctrl><H>	backspace
	<Ctrl><I>	tab
	<Ctrl><J>	linefeed
	<Ctrl><M>	carriage return

4.4.2 Editor Command Line Commands

ASCII commands	`AL ["Filename"]`	loads a source text named "filename"
	`AS ["Filename"]`	saves a source text named "filename"
Cursor movement	`CB (Cursor bottom)`	moves cursor to end of text/fold
	`CD (Cursor down)`	moves cursor one line down
	`CE (Cursor end)`	moves cursor to end of line
	`CH (Cursor home)`	moves cursor to start of line

	CL (Cursor left)	moves cursor one character left
	CN (Cursor next)	moves cursor to start of next line
	CP (Cursor previous)	moves cursor to start of previous line
	CR (Cursor right)	moves cursor one character right
	CS (Cursor start)	moves cursor to start of current line
	CT (Cursor top)	moves cursor to start of text/fold
	CU (Cursor up)	moves cursor one line up
Delete commands	DB (delete back)	backspace
	DC (delete char)	delete character
	DL (delete line)	delete line
Line commands		
	LS (line split)	divides line into two lines at current cursor position
Quit command	Q	exits editor
Set commands	Sa	Auto Indent off
	SA	Auto Indent on
	Sb	SkipBlanks off
	SB	SkipBlanks on
	Si	Insert mode
	So	Overwrite mode
	ST	Tabs on
	St	Tabs off
eXit command	X	saves changes and exits editor

Appendix A Editor listings

This section contains complete editor listings of the first version and the final version of the editor program, for users that wish to type in the editor source code. Changes in versions 2 thru 5 are outlined in the text. The optional disk available for this books contains the complete source code for all six version of the editor.

Version 1 makefile

```
.c.o:
  cc +B +Ipre/Editor.pre -O $@ src/$*.c

.prelist.pre:
  cc -A -O ram:$*.preasm +H$@ pre/$*.prelist
  delete ram:$*.preasm

OBJ=src/Editor.o

Editor: $(OBJ)
   ln $(OBJ) -lc -o Editor

Debug: $(OBJ)
   ln -w $(OBJ) -lc -o Editor

pre/Editor.pre: pre/Editor.prelist src/Editor.h
src/Editor.o: src/Editor.c src/Editor.h pre/Editor.pre
   cc +Ipre/Editor.pre src/Editor.c
```

Version 1 Editor.h

```
/************
 *
 * Includes:
 *
 ************/

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

/***********
 *
 * Defines:
 *
 ***********/

#define MAXWIDTH 80
#define EVENLEN(x)  (((x)+1)&-2)

/*******************
 *
 * Data structures:
 *
 *******************/

struct Zline
{
  struct Zline *succ;
  struct Zline *pred;
  UBYTE flags;
  UBYTE len;
};

#define ZLF_USED 128

struct ZList
{
  struct Zline *head;
  struct Zline *tail;
  struct Zline *tailpred;
};

struct Memorysection
{
  struct Memorysection *succ;
  struct Memorysection *pred;
  UWORD len;
};

struct FList
{
  struct Memorysection *head;
  struct Memorysection *tail;
  struct Memorysection *tailpred;
};

struct Memoryblock
```

```
{
  struct Memoryblock *succ;
  struct Memoryblock *pred;
  ULONG length;
  ULONG free;
  struct FList freeliste;
};

struct BList
{
  struct Memoryblock *head;
  struct Memoryblock *tail;
  struct Memoryblock *tailpred;
};

struct Editor
{
  struct Editor *succ;
  struct Editor *pred;
  struct Window *window;
  struct BList block;
  struct ZList zlines;
  UBYTE buffer[MAXWIDTH];
  UBYTE tabstring[MAXWIDTH];
  UWORD num_lines;
  struct Zline *actual;
  struct Zline *top;
  UWORD toppos;
  UWORD xpos,ypos;
  UWORD changed:1;
  UWORD insert:1;
};

struct EList
{
  struct Editor *head;
  struct Editor *tail;
  struct Editor *tailpred;
};
```

Version 1 Editor.c

```
/* Editor.c V0.1 */
/************
 * Includes:
 ************/
#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"

/***********
 * Defines:
 ***********/

#define REV 33L
#define MAXINPUTLEN 128L

/**********************
 * External Functions:
 **********************/

struct Library *OpenLibrary();
struct Window *OpenWindow();
void CloseLibrary(),NewList(),AddTail(),CloseWindow(),free();
void DeletePort(),ModifyIDCMP(),ReplyMsg();
struct Editor *malloc(),*RemHead();
struct MsgPort *CreatePort();
struct IntuiMessage *GetMsg();
ULONG Wait(),OpenDevice();
SHORT RawKeyConvert();

/*********************
 * Global Variables:
 *********************/

struct Library *IntuitionBase = NULL, *GfxBase = NULL, *DosBase = NULL;
struct Device *ConsoleDevice = NULL;

struct EList editorList;
struct Editor *ActualEditor;

struct NewWindow newEdWindow =
{
  100,40,440,156,
  AUTOFRONTPEN,AUTOBACKPEN,
  REFRESHWINDOW | MOUSEBUTTONS | RAWKEY | CLOSEWINDOW,
  WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH | WINDOWCLOSE | SIZEBBOTTOM
  | SIMPLE_REFRESH | ACTIVATE,
  NULL,NULL,
  (UBYTE *)"Editor",
  NULL,NULL,
  100,40,640,200,
  WBENCHSCREEN
};

struct MsgPort *edUserPort = NULL;

struct IOStdReq ioStdReq;
```

```
UBYTE inputBuffer[MAXINPUTLEN];
UWORD inputLen = 0;

struct InputEvent inputEvent =
{
  0,
  IECLASS_RAWKEY,0,
  0,0
};

/***************
 * Functions:
 ***************/

/**********************************
 * OpenEditor()
 * Open Editor window and
 * initalize Editor structure
 * Returns pointer to Editor structure
 * or is zero in case of an error
 **********************************/

struct Editor *OpenEditor()
{
  register struct Editor *ed = NULL;
  register struct Window *wd;
  register ULONG flags;

  /* IDCMPFlags saved, sets it to zero in structure
    => use your own UserPort! */
  flags = newEdWindow.IDCMPFlags;
  newEdWindow.IDCMPFlags = NULL;

  /* Memory for editor structure: */
  if (ed = malloc(sizeof(struct Editor)))
    /* Window open: */
    if (wd = OpenWindow(&newEdWindow))
    {
      ed->window = wd;

      /* UserPort established: */
      wd->UserPort = edUserPort;
      ModifyIDCMP(wd,flags);

      /* Parameter initialization: */
      NewList(&(ed->block));
      NewList(&(ed->zlines));
      ed->num_lines = 0;
      ed->actual = NULL;
      ed->top = NULL;
      ed->toppos = 0;
      ed->xpos = 1;
      ed->ypos = 1;
      ed->changed = 0;
      ed->insert = 1;
    }
    else
    {
      free(ed);
      ed = NULL;
    }
```

```
  newEdWindow.IDCMPFlags = flags;
  return (ed);
}

/***************************
 * CloseEditor(ed)
 * Close editor window.
 * ed ^ Editor structure.
 ***************************/

void CloseEditor(ed)
struct Editor  *ed;
{
  /* Close window: */
  ed->window->UserPort = NULL;
  CloseWindow(ed->window);
  free(ed);
}

/*****************
 * Main program:
 *****************/

main()
{
  register struct Editor *ed;
  BOOL running = TRUE;
  register struct IntuiMessage *imsg;
  ULONG signal,class;
  UWORD code,qualifier;
  APTR iaddress;
  register UWORD n = 1;
  SHORT l;

  /* Libraries open: */

  if ( !(IntuitionBase = OpenLibrary("intuition.library",REV)))
    goto Ende;
  if ( !(GfxBase = OpenLibrary("graphics.library",REV)))
    goto Ende;
  if ( !(DosBase = OpenLibrary("dos.library",REV)))
    goto Ende;
  if ( !(OpenDevice("console.device",-1L,&ioStdReq,0L)))
    ConsoleDevice = ioStdReq.io_Device;
  else
    goto Ende;

  /* UserPort open */
  if (! (edUserPort = CreatePort(NULL,0L))) goto Ende;

  /* Editor list initialization: */
  NewList(&editorList);

  /* Open first editor window: */
  if (ed = OpenEditor())
  {
    AddTail(&editorList,ed);
    ActualEditor = ed;
  }
  else
    goto Ende;
```

```
    do
    {
      signal = Wait(1L << edUserPort->mp_SigBit);

      while (imsg = GetMsg(edUserPort))
      {
        class       = imsg->Class;
        code        = imsg->Code;
        qualifier = imsg->Qualifier;
        iaddress    = imsg->IAddress;

        ReplyMsg(imsg);

        /* Event processing: */
        switch (class)
        {
          case RAWKEY:
            if (! (code & IECODE_UP_PREFIX))
            {
              inputEvent.ie_Code     = code;
              inputEvent.ie_Qualifier = qualifier;
              if ((l = RawKeyConvert(
                    &inputEvent,inputBuffer,MAXINPUTLEN,NULL)
                    ) >= 0)
              {
                inputBuffer[l] = 0;
                printf("%3d> Length = %d: ",n++,l);
                for (l = 0; inputBuffer[l]; l++)
                  printf("%2x",(UWORD)inputBuffer[l]);
                if (inputBuffer[0] >= 32 && inputBuffer[0] <= 127)
                  printf(" %s",inputBuffer);
                printf("\n");
              }
            }
            break;

          case CLOSEWINDOW:
            running = FALSE;
            break;

          default:
            printf("Not a processable event: %lx\n",class);

        } /* of case */
      } /* of while (GetMsg()) */
    } while (running);

  Ende:
    /* Close all editor windows: */
    while (ed = RemHead(&editorList))
      CloseEditor(ed);

    /* Close UserPort: */
    if (edUserPort) DeletePort(edUserPort);

    /* Close Libraries: */
    if (DosBase) CloseLibrary(DosBase);
    if (GfxBase) CloseLibrary(GfxBase);
    if (IntuitionBase) CloseLibrary(IntuitionBase);
  }
```

Version 4 TestText

The ¶ characters in the following file represent the linefeed character, decimal 10. The » character at the end of the file represents the Escape character, decimal 27. These characters and the "a" must be added to the file since redirection is used to load the test file into Version 4 of the editor for testing. We used the `replace` command from the Abacus Amiga DOS Toolbox to replace a single linefeed with "linefeed,a,linefeed" (10,97,10) and then placed an "escape,linefeed, linefeed" (27,10,10) at the end of the file.

Notice the folding marker comments in this text, make sure to add these to your `TestText` file. The redirection feature of AmigaDOS allows the a's and linefeeds in this file to be interpeted as input for this version of the editor. Remember that an Escape character (decimal 27) must be at the end of the file.

```
a¶
/* TestText for testing the editor */¶
a¶
/*#FOLD: Includes */¶
a¶
/************¶
a¶
 *¶
a¶
 * Includes:¶
a¶
 *¶
a¶
 ************/¶
a¶
¶
a¶
#include <exec/types.h>¶
a¶
#include <intuition/intuition.h>¶
a¶
#include "src/Editor.h"¶
a¶
/*#ENDFD Includes */¶
a¶
/*#FOLD: Defines */¶
a¶
/***********¶
a¶
 *¶
a¶
 * Defines:¶
a¶
 *¶
a¶
 ***********/¶
a¶
¶
```

```
a¶
#define REV 33L¶
a¶
#define MAXINPUTLEN 128L¶
a¶
/*#ENDFD Defines */¶
a¶
/*#FOLD: External Functions */¶
a¶
/**********************¶
a¶
 *¶
a¶
 * External Functions:¶
a¶
 *¶
a¶
 **********************/¶
a¶
¶
a¶
void Test(),print();¶
a¶
¶
a¶
struct Library *OpenLibrary();¶
a¶
struct Window *OpenWindow();¶
a¶
void CloseLibrary(),NewList(),AddTail(),CloseWindow(),free();¶
a¶
void DeletePort(),ModifyIDCMP(),ReplyMsg(),freeMemoryblock();¶
a¶
void initWindowSize(),SetDrMd(),SetAPen(),SetBPen(),printAll();¶
a¶
void handleKeys(),Cursor();¶
a¶
struct Editor *malloc(),*RemHead();¶
a¶
struct MsgPort *CreatePort();¶
a¶
struct IntuiMessage *GetMsg();¶
a¶
ULONG Wait(),OpenDevice();¶
a¶
SHORT RawKeyConvert();¶
a¶
¶
a¶
/*#ENDFD External Functions */¶
»¶
¶
```

Version 6 makefile

```
.c.o:
  cc +B +I/pre/Editor.pre -O $@ src/$*.c

.prelist.pre:
  cc -A -O ram:$*.preasm +H$@ /pre/$*.prelist
  delete ram:$*.preasm

OBJ=src/Editor.o src/Memory.o src/Output.o src/Cursor.o src/Edit.o\
   src/Command.o

Editor: $(OBJ)
  ln $(OBJ) -lc -o Editor

Debug: $(OBJ)
  ln -w $(OBJ) -lc -o Editor

/pre/Editor.pre: /pre/Editor.prelist src/Editor.h
src/Editor.o: src/Editor.c src/Editor.h /pre/Editor.pre
  cc +I/pre/Editor.pre src/Editor.c
src/Memory.o: src/Memory.c src/Editor.h /pre/Editor.pre
src/Output.o: src/Output.c src/Editor.h /pre/Editor.pre
src/Cursor.o: src/Cursor.c src/Editor.h /pre/Editor.pre
src/Edit.o: src/Edit.c src/Editor.h /pre/Editor.pre
src/Command.o:   src/Command.c src/Editor.h       /pre/Editor.pre
```

Version 6 Editor.h

```
/************
 *
 * Includes:
 *
 ************/

#include <stdio.h>
#include <exec/types.h>
#include  <intuition/intuitionbase.h>
#include  <intuition/intuition.h>

/***********
 *
 * Defines:
 *
 ***********/

#define MAXWIDTH 80
#define MAXHEIGHT 64
#define BGPEN 0L
#define FGPEN 1L
#define FOLDPEN 2L
#define CTRLPEN 3L
#define EVENLEN(x) (((x)+1)&-2)
#define CONTROLCODE(c) (((c)&127)<32)

#define ZLINESPTR(n) actualEditor->zlinesptr[n]
#define ZLINESNR(n) actualEditor->zlinesnr[n]

#define INFO_XPOS_INC 2
#define INFO_XPOS_LEN 4
#define INFO_YPOS_INC 9
#define INFO_YPOS_LEN 4
#define INFO_NUMOFLINES_INC 16
#define INFO_NUMOFLINES_LEN 4
#define INFO_MINFOLD_INC 22
#define INFO_MINFOLD_LEN 2
#define INFO_MAXFOLD_INC 25
#define INFO_MAXFOLD_LEN 2
#define INFO_FLAGS_INC 29
#define INFO_FLAGS_LEN 5

/******************
 *
 * Data structures:
 *
 ******************/

struct Zline
{
  struct Zline *succ;
  struct Zline *pred;
  UBYTE flags;
  UBYTE len;
};
```

```
#define ZLF_USED 128
#define ZLF_FSE 64
#define ZLF_FOLD 63

struct ZList
{
  struct Zline *head;
  struct Zline *tail;
  struct Zline *tailpred;
};

struct Memorysection
{
  struct Memorysection *succ;
  struct Memorysection *pred;
  UWORD len;
};

struct FList
{
  struct Memorysection *head;
  struct Memorysection *tail;
  struct Memorysection *tailpred;
};

struct Memoryblock
{
  struct Memoryblock *succ;
  struct Memoryblock *pred;
  ULONG length;
  ULONG free;
  struct FList freeliste;
};

struct BList
{
  struct Memoryblock *head;
  struct Memoryblock *tail;
  struct Memoryblock *tailpred;
};

struct Editor
{
  struct Editor *succ;
  struct Editor *pred;
  struct Window *window;
  struct BList block;
  struct ZList zlines;
  UBYTE buffer[MAXWIDTH + 1];
  UBYTE buflastchar;
  UWORD buflen,bufypos;
  UBYTE tabstring[MAXWIDTH];
  UWORD num_lines;
  struct Zline *actual;
  struct Zline *zlinesptr[MAXHEIGHT];
  UWORD zlinesnr[MAXHEIGHT];
  UWORD leftpos;
  UWORD xpos,ypos;
  UWORD wdy;
  UWORD changed    : 1;
  UWORD insert     : 1;
  UWORD tabs       : 1;
```

```
  UWORD skipblanks  : 1;
  UWORD autoindent  : 1;
  struct RastPort *rp;
  UWORD xoff,yoff;
  UWORD xscr,yscr;
  UWORD cw,ch;
  UWORD wcw,wch;
  UWORD xrl,xrr,yru,bl;
  UWORD minfold;
  UWORD maxfold;
  UBYTE gc_SIBuffer[256];
  struct StringInfo gc_GadgetSI;
  struct Gadget G_Command;
  UBYTE gi_Text[36];
  struct IntuiText gi_IText;
  struct Gadget G_Info;
  ULONG ip_xpos;
  ULONG ip_ypos;
  ULONG ip_numzlines;
  ULONG ip_minfold;
  ULONG ip_maxfold;
  ULONG ip_flags;
  ULONG ip_topedge;
  UBYTE filename[80];
  UWORD rm;
  struct Zline *lineptr;
};

struct EList
{
  struct Editor *head;
  struct Editor *tail;
  struct Editor *tailpred;
};
```

Version 6 Editor.c

```
/*********************************************
 *  Editor
 *  Text editor for book Adv_C.
 *  © 1988 DATA   BECKER    ABACUS
 *********************************************/
/*#FOLD:  Includes  */
/************
 * Includes:
 ************/

#include <exec/types.h>
#include  <intuition/intuitionbase.h>
#include  <intuition/intuition.h>
#include  "src/Editor.h"
/*#ENDFD*/
/*#FOLD:  Defines */
/***********
 *
 * Defines:
 *
 ***********/

#define REV 33L
#define MAXINPUTLEN 128L
#define UT1 "USAGE: Editor [Flags] <Filename>"
#define UT2 "       Flags: [-a][-A][-b][-B][- i|I][- o|O][- r|R[=n]][-t][-
T]"
/*#ENDFD*/
/*#FOLD:  External Functions */
/**********************
 *
 * External Functions:
 *
 **********************/

struct Library *OpenLibrary();
struct Window *OpenWindow();
void CloseLibrary(),NewList(),AddTail(),CloseWindow(),free();
void DeletePort(),ModifyIDCMP(),ReplyMsg(),freeMemoryblock();
void initWindowSize(),SetDrMd(),SetAPen(),SetBPen(),printAll();
void Cursor(),saveIfCursorMoved(),strncpy(),CloseFont(),puts();
void printXpos(),printYpos(),printInfo(),DisplayBeep();
void BeginRefresh(),EndRefresh();
struct Editor *malloc(),*RemHead();
struct MsgPort *CreatePort();
struct IntuiMessage *GetMsg();
ULONG Wait(),OpenDevice();
SHORT RawKeyConvert();
struct TextFont *OpenFont();
BOOL handleKeys(),ActivateGadget(),loadASCII();
UBYTE *executeCommand(),*commandSet();
/*#ENDFD*/
/*#FOLD:  Global Variables  */
/*********************
 * Global Variables:
 *********************/
```

```
struct Library *IntuitionBase = NULL, *GfxBase = NULL, *DosBase = NULL;
struct Device *ConsoleDevice = NULL;

struct EList editorList;
struct Editor *actualEditor;

/*#FOLD: Window und Gadgets */
UBYTE gc_SIBuffer[256];
UBYTE gc_UndoBuffer[256];
struct StringInfo gc_GadgetSI =
{
  gc_SIBuffer,
  gc_UndoBuffer,
  0,
  256,
  0,
  0,0,0,0,0,
  0,
  NULL,
  NULL
};

SHORT gc_BorderVectors[4] = {0,0,350,0};
struct Border gc_Border =
{
  -1,-1,
  1,0,JAM1,
  2,
  gc_BorderVectors,
  NULL
};

struct Gadget G_Command =
{
  NULL,
  2,-9,
  -302,9,
  GADGHCOMP | GRELBOTTOM | GRELWIDTH,
  RELVERIFY | BOTTOMBORDER,
  STRGADGET,
  (APTR)&gc_Border,
  NULL,
  NULL,
  0,
  (APTR)&gc_GadgetSI,
  2,
  NULL
};

SHORT gi_BorderVectors[14] = {-1,0,-1,10,0,10,0,0,298,0,298,1,283,1};
struct Border gi_Border =
{
  0,0,
  1,0,JAM1,
  7,
  gi_BorderVectors,
  NULL
};

struct TextAttr TOPAZ80 =
{
```

```
  (STRPTR)"topaz.font",
  TOPAZ_EIGHTY,0,0
};

UBYTE gi_Text[36] = "X=   1 Y=   1 #=   0 [ 0, 0] IcATB";
struct IntuiText gi_IText =
{
  1,0,JAM2,
  4,2,
  &TOPAZ80,
  gi_Text,
  NULL
};

struct Gadget G_Info =
{
  &G_Command,
  -298,-10,
  280,10,
  GADGHNONE | GRELBOTTOM | GRELRIGHT,
  RELVERIFY | BOTTOMBORDER,
  BOOLGADGET,
  (APTR)&gi_Border,
  NULL,
  &gi_IText,
  0,
  NULL,
  1,
  NULL
};

struct NewWindow newEdWindow =
{
  0,0,640,200,
  AUTOFRONTPEN,AUTOBACKPEN,
  REFRESHWINDOW | MOUSEBUTTONS | RAWKEY | CLOSEWINDOW | NEWSIZE | GADGETUP,
  WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH | WINDOWCLOSE | SIZEBBOTTOM
  | SIMPLE_REFRESH | ACTIVATE,
  NULL,NULL,
  NULL,
  NULL,NULL,
  320,50,640,200,
  WBENCHSCREEN
};

struct TextFont *infoFont = NULL;
/*#ENDFD*/

struct MsgPort *edUserPort = NULL;

struct IOStdReq ioStdReq;

UBYTE inputBuffer[MAXINPUTLEN];
UWORD inputLen = 0;

struct InputEvent inputEvent =
{
  0,
  IECLASS_RAWKEY,0,
  0,0
};
/*#ENDFD*/
```

```
/*#FOLD:  Functions */
/*#FOLD:  nextLine  */
/*******************************************
 *
 * nextLine(line,pnr)
 *
 * Returns pointer to next line,
 * and executes folding
 * Zero, if no next line.
 *
 * line ^ line-structure.
 * pnr ^ to linenumber.
 *
 *******************************************/

struct Zline *nextLine(line,pnr)
register struct Zline *line;
register UWORD          *pnr;
{
  register struct Zline *z;
  register UWORD minfold = actualEditor->minfold;
  register UWORD maxfold = actualEditor->maxfold;

  if (z = line)
    if (z->succ)
      do
      {
        if ((z = z->succ)->succ)
          if ((z->flags & ZLF_FOLD) < minfold)
          {
            /* Following lines lie outside of fold */
            z = NULL;
            break;
          }
          else
            *pnr += 1;
        else
        {
          /* no following line */
          z = NULL;
          break;
        }
      }
      while ((z->flags & ZLF_FOLD) > maxfold);
    else
      z = NULL;

  return (z);
}
/*#ENDFD*/
/*#FOLD:  prevLine  */
/*******************************************
 *
 * prevLine(line,pnr)
 *
 * Returns pointer to previous line,
 * and executes folding
 * Zero, if no previous line.
 *
 * line ^ Zline-structure.
 * pnr ^ to line number.
 *
```

```
 ****************************************/

struct Zline *prevLine(line,pnr)
register struct Zline *line;
register UWORD          *pnr;
{
  register struct Zline *z;
  register UWORD minfold = actualEditor->minfold;
  register UWORD maxfold = actualEditor->maxfold;

  if (z = line)
    if (z->pred)
      do
      {
        if ((z = z->pred)->pred)
          if ((z->flags & ZLF_FOLD) < minfold)
          {
            /* previous line lies outside fold */
            z = NULL;
            break;
          }
          else
            *pnr -= 1;
        else
        {
          /* no previous line */
          z = NULL;
          break;
        }
      }
      while ((z->flags & ZLF_FOLD) > maxfold);
    else
      z = NULL;

  return (z);
}
/*#ENDFD*/
/*#FOLD:  OpenEditor */
/***********************************
 *
 * OpenEditor()
 *
 * Open Editor window and
 * initalize Editor structure
 *
 * Returns pointer to Editor structure
 * or is zero in case of an error
 *
 ***********************************/

struct Editor *OpenEditor()
{
  register struct Editor *ed = NULL;
  register struct Window *wd;
  register struct RastPort *rp;
  ULONG flags;

  /* IDCMPFlags saved, sets it to zero in structure
    => use your own UserPort! */
  flags = newEdWindow.IDCMPFlags;
  newEdWindow.IDCMPFlags = NULL;
```

```
/* Memory for editor structure: */
if (ed = malloc(sizeof(struct Editor)))
{
  /* Gadgets initialization: */
  ed->gc_SIBuffer[0]  = 0;
  ed->gc_GadgetSI     = gc_GadgetSI;
  ed->G_Command       = G_Command;
  ed->gi_IText        = gi_IText;
  ed->G_Info          = G_Info;
  strncpy(ed->gi_Text,gi_Text,sizeof(ed->gi_Text));

  /* set pointer: */
  ed->gc_GadgetSI.Buffer     = ed->gc_SIBuffer;
  ed->G_Command.SpecialInfo = (APTR) &(ed->gc_GadgetSI);
  ed->gi_IText.IText        = ed->gi_Text;
  ed->G_Info.NextGadget     = &(ed->G_Command);
  ed->G_Info.GadgetText     = &(ed->gi_IText);
  newEdWindow.FirstGadget    = &(ed->G_Info);
  newEdWindow.Title         = ed->filename;
  ed->filename[0]         = 0;

  /* Window open: */
  if (wd = OpenWindow(&newEdWindow))
  {
    ed->window = wd;
    ed->rp = (rp = wd->RPort);

    /* Write mode set: */
    SetDrMd(rp,JAM2);
    SetAPen(rp,FGPEN);
    SetBPen(rp,BGPEN);

    /* UserPort established: */
    wd->UserPort = edUserPort;
    ModifyIDCMP(wd,flags);

    /* Parameter initialization: */
    NewList(&(ed->block));
    NewList(&(ed->zlines));

    ed->num_lines  = 0;
    ed->actual     = NULL;
    ed->buflen     = 0;
    ed->bufypos    = 0;
    ed->buflastchar= ' ';
    ed->leftpos    = 0;
    ed->xpos       = 1;
    ed->ypos       = 1;
    ed->wdy        = 0;
    ed->changed    = 0;
    ed->insert     = 1;
    ed->tabs       = 1;
    ed->skipblanks  = 1;
    ed->autoindent  = 1;
    ed->minfold    = 0;
    ed->maxfold    = 0;
    ed->rm       = MAXWIDTH;

    {
      register UBYTE *ptr;
      register struct Zline **zptr;
      register UWORD n,*pnr;
```

```
        /* zlinesptr/nr-Array initialization: */
        for (n = 0, zptr = ed->zlinesptr, pnr = ed->zlinesnr;
            n < MAXHEIGHT; n++, zptr++, pnr++)
        {
          *zptr = NULL;
          *pnr  = 1;
        }

        /* Tab initialization: */
        ptr = ed->tabstring;
        *ptr++ = 1;
        for (n = 1; n < MAXWIDTH; n++)
          if (n % 3)
            *ptr++ = 1;
          else
            *ptr++ = 0;
      }

      {
        register struct Editor *oldae;

        ed->wch = 0;
        initWindowSize(ed);

        oldae        = actualEditor;
        actualEditor = ed;
        printInfo();
        actualEditor = oldae;
      }
    }
    else
    {
      free(ed);
      ed = NULL;
    }
  }

  newEdWindow.IDCMPFlags = flags;
  return (ed);
}
/*#ENDFD*/
/*#FOLD:  CloseEditor */
/***************************
 *
 * CloseEditor(ed)
 *
 * Close editor window.
 *
 * ed ^ Editor structure.
 *
 ***************************/

void CloseEditor(ed)
struct Editor  *ed;
{
  register struct Memoryblock *blk;

  /* Free first memory block: */
  while (blk = (struct Memoryblock *)RemHead(&(ed->block)))
    freeMemoryblock(blk);
```

```
  /* Then Close window: */
  ed->window->UserPort = NULL;
  CloseWindow(ed->window);
  free(ed);
}
/*#ENDFD*/
/*#ENDFD*/
/*#FOLD:  Main program */
/*****************.
 *
 * Main program:
 *
 *****************/

main(argc,argv)
int  argc;
UBYTE   *argv[];
{
  register struct Editor *ed;
  BOOL running = TRUE;
  register struct IntuiMessage *imsg;
  ULONG signal,class,mouseX,mouseY;
  UWORD code,qualifier;
  APTR iaddress;

  /*  Was Editor started with "Editor ?" , the Text output */
  if  (argc ==  2)
    if  (*argv[1] == '?')
    {
      puts(UT1);
      puts(UT2);
      goto Ende;
    }

  /* Libraries open: */
  if ( !(IntuitionBase = OpenLibrary("intuition.library",REV)))
    goto Ende;
  if ( !(GfxBase = OpenLibrary("graphics.library",REV)))
    goto Ende;
  if ( !(DosBase = OpenLibrary("dos.library",REV)))
    goto Ende;
  if ( !(OpenDevice("console.device",-1L,&ioStdReq,0L)))
    ConsoleDevice = ioStdReq.io_Device;
  else
    goto Ende;

  /* Font for Infoline open: */
  if (! (infoFont = OpenFont(&TOPAZ80))) goto Ende;

  /* UserPort open */
  if (! (edUserPort = CreatePort(NULL,0L))) goto Ende;

  /*  Max height for window calculated: */
  newEdWindow.Height =  newEdWindow.MaxHeight =
        ((struct  IntuitionBase *)IntuitionBase)->ActiveScreen->Height;

  /* Editor list initialization: */
  NewList(&editorList);

  /* Open first editor window: */
  if (ed = OpenEditor())
  {
```

```
    AddTail(&editorList,ed);
    actualEditor = ed;
  }
  else
    goto Ende;

  /* Command line reading: */
  if (argc >= 2)
  {
    register UWORD n = 1;

    while   (n  < argc)
    {
      if  (*argv[n] == '-')
      {
        if  (commandSet(argv[n] + 1) >= 0x80000000)
        {
          DisplayBeep(NULL);
          break;
        }
      }
      else
      {
        if  (loadASCII(argv[n]))
          printAll();
        else
          DisplayBeep(NULL);

        /*  File name end input! */
        n++;
        break;
      }

      n++;
    }

    if (n < argc)
    {
      /*  Error! */
      puts(UT1);
      puts(UT2);
      goto Ende;
    }
  }

  /*#FOLD:  Main loop */
  do
  {
    /* Set cursor: */
    Cursor();

    signal = Wait(1L << edUserPort->mp_SigBit);

    /* erase cursor again: */
    Cursor();

    while (imsg = GetMsg(edUserPort))
    {
      class       = imsg->Class;
      code        = imsg->Code;
      qualifier   = imsg->Qualifier;
      iaddress    = imsg->IAddress;
```

```
mouseX     = imsg->MouseX;
mouseY     = imsg->MouseY;

ReplyMsg(imsg);

/* Event processing: */
switch (class)
{
  case RAWKEY:
  {
    register struct IntuiMessage *im1,*im2;

    if (! (code & IECODE_UP_PREFIX))
    {
      inputEvent.ie_Code     = code;
      inputEvent.ie_Qualifier = qualifier;
      if ((inputLen = RawKeyConvert(
            &inputEvent,inputBuffer,MAXINPUTLEN,NULL)
            ) >= 0)
        running = handleKeys(inputBuffer,inputLen);
    }

    /* run-on suppressed: */
    im1 = (struct IntuiMessage *)
        edUserPort->mp_MsgList.lh_Head;
    while (im2 = (struct IntuiMessage *)
             im1->ExecMessage.mn_Node.ln_Succ)
    {
      if (im2->ExecMessage.mn_Node.ln_Succ == NULL) break;
      if (im1->Class != RAWKEY) break;
      if (!(im1->Qualifier & IEQUALIFIER_REPEAT)) break;
      if (im2->Class != RAWKEY) break;
      if (!(im2->Qualifier & IEQUALIFIER_REPEAT)) break;

      /* Message reply: */
      im1 = GetMsg(edUserPort);
      ReplyMsg(im1);

      im1 = (struct IntuiMessage *)
          edUserPort->mp_MsgList.lh_Head;
    }

    break;
  }
  case MOUSEBUTTONS:
    {
      register WORD x,y;

      if (mouseX <= actualEditor->xoff)
        x = 0;
      else
        x = (mouseX - actualEditor->xoff)
          / actualEditor->cw;
      if (++x >= actualEditor->wcw)
        x = actualEditor->wcw - 1;
      x += actualEditor->leftpos;
      if (x > MAXWIDTH)
        x = MAXWIDTH;

      if (mouseY <= actualEditor->yoff)
        y = 0;
      else
```

```
        y = (mouseY - actualEditor->yoff)
          / actualEditor->ch;

      if ((y < actualEditor->wch)
        && (actualEditor->zlinesptr[y]))
      {
        actualEditor->xpos = x;
        actualEditor->wdy  = y;
        actualEditor->ypos = actualEditor->zlinesnr[y];

        printXpos();
        printYpos();
      }
    }

  case NEWSIZE:
    initWindowSize(actualEditor);

    /* check if cursor still in window! */
    if (actualEditor->xpos > actualEditor->leftpos
                      +actualEditor->wcw)
    {
      actualEditor->xpos = actualEditor->leftpos
                     + actualEditor->wcw;
      printXpos();
    }

    if (actualEditor->wdy >= actualEditor->wch)
    {
      actualEditor->ypos = actualEditor->zlinesnr
        [(actualEditor->wdy  = actualEditor->wch - 1)];
      printYpos();
    }

    break;

  case REFRESHWINDOW:
    BeginRefresh(actualEditor->window);
    printAll();
    EndRefresh(actualEditor->window,TRUE);
    break;

  case GADGETUP:
  {
    register UBYTE *ptr;
    register SHORT pos,hw;

    if (ptr = executeCommand(actualEditor->gc_SIBuffer))
      if (ptr == -1L)
        running = FALSE;
      else
      {
        if   (ptr >= 0x80000000)
          ptr =   NULL - ptr;

        /* set cursor in StringGadget to error */
        pos = ptr - actualEditor->gc_SIBuffer;
        actualEditor->gc_GadgetSI.BufferPos = pos;
        if (pos < actualEditor->gc_GadgetSI.DispCount)
          actualEditor->gc_GadgetSI.DispPos = 0;
        else
          actualEditor->gc_GadgetSI.DispPos = pos
```

```
                    - actualEditor->gc_GadgetSI.DispCount / 2;

                /* Gagdet aktivated: */
                ActivateGadget(&(actualEditor->G_Command),
                            actualEditor->window,NULL);
                DisplayBeep(NULL);
              }

            break;
          }

          case CLOSEWINDOW:
            running = FALSE;
            break;

          default:
            printf("Not a processable event: %lx\n",class);

        } /* of case */

        saveIfCursorMoved();
      } /* of while (GetMsg()) */
    } while (running);
    /*#ENDFD*/

  Ende:
    /* Close all editor windows: */
    while (ed = RemHead(&editorList))
      CloseEditor(ed);

    /* Close UserPort: */
    if (edUserPort)
    {
      DeletePort(edUserPort);
      edUserPort = NULL;
    }

    /* Font closed: */
    if (infoFont)
    {
      CloseFont(infoFont);
      infoFont = NULL;
    }

    /* Close Libraries: */
    if (DosBase)
    {
      CloseLibrary(DosBase);
      DosBase = NULL;
    }
    if (GfxBase)
    {
      CloseLibrary(GfxBase);
      GfxBase = NULL;
    }
    if (IntuitionBase)
    {
      CloseLibrary(IntuitionBase);
      IntuitionBase = NULL;
    }
  }
  /*#ENDFD*/
```

Version 6 Edit.c

```
/****************************************
 *
 *  Module: Edit
 *  Functions to edit the text
 *
 ****************************************/

/*#FOLD:  Includes  */
/************
 * Includes:
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"
/*#ENDFD*/
/*#FOLD:  Defines */
/************
 * Defines:
 ************/

#define CR 13
#define LF 10
#define TAB 9

/* Lenght of fold start and end: */
#define FSE_LEN 10
/* Number of characters, significant for fold marking: */
#define FSE_SIG 8
/*#ENDFD*/
/*#FOLD:  External Functions */
/**********************
 * External Functions:
 **********************/

UWORD convertLineForPrint(),strncmp(),recalcTopOfWindow();
void deleteZline(),Insert(),printAll(),restoreZlinenptr();
void DisplayBeep(),printLine(),AddHead(),scrollRight(),scrollUp();
void ScrollRaster(),printXpos(),printYpos(),printFold();
void printNumLines(),printFlags();
struct Zline *newZline(),*nextLine(),*prevLine();
BOOL cursorLeft(),cursorRight(),cursorDown(),cursorHome(),cursorEnd();
/*#ENDFD*/
/*#FOLD:  External Variables  */
/**********************
 * External Variables:
 **********************/

extern struct Editor *actualEditor;
/*#ENDFD*/
/*#FOLD:  Global Variables  */
/**********************
 * Global Variables:
 **********************/

UBYTE foldStart[] = "/*#FOLD:*/";
```

```
UBYTE foldEnde[]    = "/*#ENDFD*/";
/*#ENDFD*/

/***************
 * Functions:
 ***************/

/*#FOLD:  getLineForEdit  */
/*******************************
 * getLineForEdit(line,znr)
 * Copy line to edit into
 * the buffer.
 * line ^ Zline.
 * znr = line number of the line.
 *******************************/

void getLineForEdit   (line,znr)
register struct Zline *line;
register UWORD            znr;
{
  register UWORD leftpos;

  /* get leftpos: */
  leftpos = actualEditor->leftpos;
  actualEditor->leftpos = 0;

  /* convert line: */
  actualEditor->buflen =
    convertLineForPrint(line+1,line->len,MAXWIDTH + 1,
      actualEditor->buffer,&actualEditor->buflastchar);

  /* mark line as  "used"  : */
  line->flags |= ZLF_USED;
  actualEditor->actual = line;
  actualEditor->bufypos = znr;

  /* establish Leftpos: */
  actualEditor->leftpos = leftpos;
}
/*#ENDFD*/
/*#FOLD:  cursorOnText */
/*******************************
 * cursorOnText:
 * make sure that the cursor
 * is found on text.
 *******************************/

void cursorOnText()
{
  register struct Zline **zptr;
  register UWORD l;

  /* makes sure that the cursor is really on text: */
  zptr = &(ZLINESPTR(l = actualEditor->wdy));
  while ((*zptr == NULL) && l)
  {
    zptr--;
    l--;
  }
  actualEditor->wdy  = l;
  actualEditor->ypos = ZLINESNR(l);
}
```

```
/*#ENDFD*/
/*#FOLD:  deconvertLine */
/*************************
 * deconvertLine():
 * Deconverts line.
 * buf  ^ target buffer.
 * buffer^ source pointer.
 * buflen= buffer length.
 *************************/

UWORD deconvertLine(buf, buffer,buflen)
UBYTE          *buf,*buffer;
UWORD                  buflen;
{
  register UBYTE *p1,*p2,*tab,*fb = NULL;
  register WORD l;

  /* deconvert line: */
  p1 = buffer;
  p2 = buf;
  tab= actualEditor->tabstring;
  l   = buflen;

  if (actualEditor->tabs)
    /* Spaces converted to tabs: */
    while (l)
    {
      if ((*p2 = *p1++) == ' ')
      {
        if (fb == NULL)
          fb = p2;
      }
      else
      {
        if (fb)
          fb = NULL;
      }

      /* fb points to first space */
      /* other characters are copied. (first blank)      */
      if ((! *++tab) && (fb))
      {
        *fb = TAB;
        fb++;
        p2 = fb;
      }
      else
        p2++;

      l--;
    }
  else
    while (l)
    {
      *p2 = *p1;
      p1++;
      p2++;
      l--;
    }

  /* Spaces at end deleted: */
  if (actualEditor->skipblanks)
```

```
    while ((p2 > buf) && ((*(p2-1) == ' ') || (*(p2-1) == TAB)))
      p2--;

  return ((UWORD)(p2 - buf));
}
/*#ENDFD*/
/*#FOLD:  getFoldInc */
/****************************************
 * getFoldInc(line)
 * Tests if line is a fold start
 * or end mark and gives 1 or -1
 * back.
 * Else a 0 is returned.
 * line ^ corresponding line.
 ****************************************/

UWORD getFoldInc(line)
struct Zline   *line;
{
  register UBYTE *p1;
  register UWORD l;

  /* Fold marker at beginning of line! */
  p1 = (UBYTE *)(line + 1);
  l  = line->len;
  while (l && ((*p1 == ' ') || (*p1 == TAB)))
  {
    p1++;
    l--;
  }

  /* Type of fold markers (-> l): */
  /*  1 = Start,                          */
  /*  -1   End and                                */
  /*  0 no fold marker.           */
  if (l >= FSE_SIG)
    if (strncmp(p1,foldStart,FSE_SIG) == 0)
      l = 1;
    else if (strncmp(p1,foldEnde,FSE_SIG) == 0)
      l = -1;
    else
      l = 0;
  else
    l = 0;  /* has no fold markers */

  return (l);
}
/*#ENDFD*/
/*#FOLD:  saveLine  */
/*******************************
 * saveLine(line)
 * Saves line again.
 * line ^ old line structure.
 *******************************/

BOOL saveLine(line)
struct Zline *line;
{
  static UBYTE buf[MAXWIDTH + 2];       /* +2 fuer CR und LF */
  register UBYTE *p1,*p2,*tab,*fb = NULL;
  register WORD l,len;
  struct Zline *pred,*old,**zptr;
```

```
/* Only for security: */
if (actualEditor->actual == NULL)
  return (FALSE);

/* deconvert line: */
p2 = buf + (len = deconvertLine(buf,actualEditor->buffer,
                        actualEditor->buflen));

/* determine if line ended with CR or LF, */
/*  and increases len correspondingly. */
p2 = buf + len;
if (l = line->len)
{
  p1 = ((UBYTE *)(line + 1)) + line->len - 1;
  if (*p1 == CR)
  {
    len++;
    *p2 = CR;
  }
  else if (*p1 == LF)
  {
    len++;
    if ((l > 1) && (*--p1 == CR))
    {
      len++;
      *p2++ = CR;
    }
    *p2 = LF;
  }
}

/* use new line: */
if (EVENLEN(len) != EVENLEN(line->len))
{
  actualEditor->lineptr = line;        /* Garbage-Collection    */
  if (line = newZline(len))
  {
    old = actualEditor->lineptr;
    Insert(&actualEditor->zlines,line,old);
    line->flags = old->flags & ~ZLF_USED;
    deleteZline(old);

    /* now possible to convert pointer: */
    for (l = 0, zptr = actualEditor->zlinesptr;
        l <= actualEditor->wch; l++, zptr++)
      if (*zptr == old)
      {
        *zptr = line;
        break;
      }

    actualEditor->lineptr = NULL;
  }
  else
  {
    actualEditor->lineptr = NULL;
    return (FALSE);
  }
}
else
{
```

```
  line->len   = len;
  line->flags &= ~ZLF_USED;
}

/* write line back: */
p1 = (UBYTE *)(line + 1);
p2 = buf;
l  = len;

while (l)
{
  *p1 = *p2;
  p1++;
  p2++;
  l--;
}

/* reconstruct folding: */
/* was old or is new line a fold marker? */
l = getFoldInc(line);
if ((line->flags & ZLF_FSE) || l)
{
  /* next the ZLF_FSE  flag set to 1: */
  if (l)
    line->flags |= ZLF_FSE;
  else
    line->flags &= ~ZLF_FSE;

  /* calculate how the fold level of the following lines */
  /* must be changed: level += l.                        */
  if ((old = line->succ)->succ)
  {
    if (((line->flags+1) & ZLF_FOLD) == (old->flags & ZLF_FOLD))
      /* Old line was fold start: */
      l -= 1;
    else if ((line->flags & ZLF_FOLD) == ((old->flags+1) & ZLF_FOLD))
      /* Old line was fold end: */
      l += 1;

    if (l)
    {
      /* change fold level of following lines: */
      while (old->succ)
      {
        old->flags = (old->flags & ~ZLF_FOLD)
                | ((old->flags + l) & ZLF_FOLD);
        old = old->succ;
      }

      restoreZlinenptr();
      printAll();
      cursorOnText();
      printYpos();
    }
  }
}

/* Display lines for security: */
/* First calculate Ypos:       */
for (l = 0, zptr = actualEditor->zlinesptr;
     l <= actualEditor->wch; l++, zptr++)
  if (*zptr == line)
```

```
    {
      printLine(line,actualEditor->yoff + actualEditor->ch*l);
      break;
    }

  /* Delete reference: */
  actualEditor->actual = NULL;
  actualEditor->buflen   = 0;
  actualEditor->bufypos = 0;

  /* Text was changed! */
  actualEditor->changed = 1;
  printFlags();

  return (TRUE);
}
/*#ENDFD*/
/*#FOLD:  saveIfCursorMoved */
/*************************************
 * saveIfCursorMoved:
 * Saves line in buffer,
 * if the cursor is moved.
 *************************************/

void saveIfCursorMoved()
{
  register UWORD bufypos;

  if (bufypos = actualEditor->bufypos)
    if (actualEditor->ypos != bufypos)
      if (! saveLine(actualEditor->actual))
        DisplayBeep(NULL);
}
/*#ENDFD*/
/*#FOLD:  undoLine  */
/*****************************************
 * undoLine:
 * Makes changes in actual line
 * before cursor leaves line
 *****************************************/

void undoLine()
{
  register struct Zline *z;

  if (z = actualEditor->actual)
  {
    actualEditor->actual = NULL;
    actualEditor->bufypos = 0;
    actualEditor->buflen  = 0;

    z->flags &= ~ZLF_USED;

    /* Line output: */
    printLine(z,actualEditor->yoff
            + actualEditor->ch*actualEditor->wdy);
  }
}
/*#ENDFD*/
/*#FOLD:  getBufferPointer */
/*************************************************
 * getBufferPointer(pptr,pinc,prest):
```

```
 * Initialize pointer to actualEditor->buffer.
 * The biffer is initialized.
 * FALSE is returned if the buffer
 * cannot be initialized.
 * pptr ^ Pointer in buffer at cursor position.
 * pinc ^ Offset to buffer start.
 * prest^ buflen - inc.
 ************************************************/

BOOL getBufferPointer(pptr,pinc, prest)
UBYTE           **pptr;
UWORD               *pinc,*prest;
{
  register UBYTE *ptr;
  register WORD inc,rest;

  /* Buffer initialized: */
  if (actualEditor->actual == NULL)
  {
    register UWORD wdy;

    if (ZLINESPTR(wdy = actualEditor->wdy))
      getLineForEdit(ZLINESPTR(wdy),ZLINESNR(wdy));
    else
      if (actualEditor->num_lines == 0)
      {
        /* create first line: */
        register struct Zline *z;

        if (z = newZline((UWORD) 1))
        {
          *((UBYTE *)(z + 1)) = LF;
          AddHead(&actualEditor->zlines,z);
          ZLINESPTR(0) = z;
          ZLINESNR(0)  = (actualEditor->num_lines = 1);
          getLineForEdit(ZLINESPTR(0),ZLINESNR(0));
          printNumLines();
        }
        else
          return (FALSE);
      }
      else
        /* Cursor is not in line! */
        return (FALSE);
  }

  /* Calculate pointer to cursor position: */
  inc = actualEditor->xpos - 1;
  ptr = actualEditor->buffer + inc;
  rest= actualEditor->buflen - inc;

  /* What if inc > buflen? */
  if (inc < MAXWIDTH)
  {
    if (rest < 0)
    {
      register UBYTE *p;

      p = actualEditor->buffer + actualEditor->buflen;
      actualEditor->buflen -= rest;
      while (rest)
      {
```

```
            *p = ' ';
            p++;
            rest++;
          }
        }
      }
      else
        return (FALSE);

      *pptr = ptr;
      *pinc = inc;
      *prest= rest;
      return (TRUE);
}
/*#ENDFD*/
/*#FOLD:  deleteChar */
/************************************
 * deleteChar:
 * Delet echaracter under the cursor.
 * False is returned if no more
 * characters.
 ************************************/

BOOL deleteChar()
{
  UBYTE *ptr;
  UWORD inc,rest;
  register UBYTE *p1,*p2;
  register UWORD l;

  if (! getBufferPointer(&ptr,&inc,&rest))
    return (FALSE);

  if (rest)
  {
    p1 = ptr + 1;
    p2 = ptr;
    l = --rest;
    while (l)
    {
      *p2 = *p1;
      p1++;
      p2++;
      l--;
    }
    *p2 = actualEditor->buflastchar;

    /* Lenght - 1 */
    actualEditor->buflen--;

    printLine(actualEditor->actual,actualEditor->yoff
              + actualEditor->ch*actualEditor->wdy);
  }

  return (TRUE);
}
/*#ENDFD*/
/*#FOLD:  backspaceChar */
/******************************
 * backspaceChar:
 * Delete character before cursor.
 * False is returned when
```

```
 * cursor is at start of line.
 ******************************/

BOOL backspaceChar()
{
  UBYTE *ptr;
  UWORD inc,rest;
  register UBYTE *p1,*p2;
  register UWORD l;

  if (! getBufferPointer(&ptr,&inc,&rest))
    return (FALSE);

  if (inc)
  {
    if (actualEditor->insert)
    {
      p1 = ptr;
      p2 = --ptr;
      l = rest;
      while (l)
      {
        *p2 = *p1;
        p1++;
        p2++;
        l--;
      }
      *p2 = actualEditor->buflastchar;

      /* Lenght - 1 */
      actualEditor->buflen--;
      inc--;
    }
    else
    {
      *--ptr = ' ';
      inc--;
      rest++;
    }

    cursorLeft();
    printLine(actualEditor->actual,actualEditor->yoff
              + actualEditor->ch*actualEditor->wdy);
  }

  return (TRUE);
}
/*#ENDFD*/
/*#FOLD:  insertLine */
/**************************
 * insertLine(c):
 * Inserts new line behind
 * actual line.
 * FALSE returned if
 * error encountered.
 * c = line end (CR/LF).
 *************************/

BOOL insertLine(c)
UBYTE          c;
{
  static UBYTE buf[MAXWIDTH + 1];
```

```
UBYTE *ptr;
register UBYTE *p1,*p2;
WORD inc,rest,len;
UWORD autoindent;
register UWORD l;
register struct Zline *z;

/* Insert new line only if actual line
  has no fold markers!!! */
if ((z = actualEditor->actual) == NULL)
  z = ZLINESPTR(actualEditor->wdy);

if (!z || !(z->flags & ZLF_FSE))
{
  /* get line in buffer: */
  if (! getBufferPointer(&ptr,&inc,&rest))
    return (FALSE);

  /* calculate auto indent: */
  if (actualEditor->autoindent)
  {
    p1 = actualEditor->buffer;
    l   = inc;
    while (l && ((*p1 == ' ') || (*p1 == TAB)))
    {
      p1++;
      l--;
    }

    if (l)
      autoindent = p1 - actualEditor->buffer + 1;
    else
      /* line consisits only of spaces: */
      autoindent = 1;
  }
  else
    autoindent = 1;

  /* deconvert line: */
  p2 = buf + (len = deconvertLine(buf,actualEditor->buffer,inc));
  if (*p2 = c)
    len++;

  if (z = newZline(len))
  {
    z->flags = (actualEditor->actual->flags & ~ZLF_USED);

    /* save line that the cursor is in: */
    p1 = (UBYTE *)(z + 1);
    p2 = buf;
    l = len;

    while (l)
    {
      *p1 = *p2;
      p1++;
      p2++;
      l--;
    }

    /* bind line: */
    Insert(&actualEditor->zlines,z,actualEditor->actual->pred);
```

```
actualEditor->num_lines++;
actualEditor->changed = 1;

/* move rest of buffer line in front: */
/* remember auto indent! */
p1 = actualEditor->buffer;
l  = autoindent;
while (--l)
  p1++;       /* Old contents remain! */

p2 = ptr;
l = rest;

if  (actualEditor->autoindent)
  /*  Spaces at start cot off:    */
  while (l  &&  (*p2 == ' '))
  {
    p2++;
    l--;
  }

/*  The write rest of the buffer start:   */
while (l)
{
  *p1 = *p2;
  p1++;
  p2++;
  l--;
}
actualEditor->buflen = p1 - actualEditor->buffer;

/* rest filled with lastchar: */
l = inc + 1 - autoindent;
while (l)
{
  *p1 = actualEditor->buflastchar;
  p1++;
  l--;
}

/* determine if new line is a fold marker: */
if (l = getFoldInc(z))
{
  register struct Zline *old;
  register UWORD fold;

  z->flags |= ZLF_FSE;

  /* change fold level of following lines: */
  if ((old = z->succ)->succ)
    while (old->succ)
    {
      old->flags = (old->flags & ~ZLF_FOLD)
              | ((old->flags + l) & ZLF_FOLD);
      old = old->succ;
    }

  /* fit minfold and maxfold: */
  if ((l > 0) || ((z->flags & ZLF_FOLD) > 0))
  {
    fold = (z->flags & ZLF_FOLD) + l;
    if (actualEditor->minfold > fold)
```

```
    {
      actualEditor->minfold = fold;
      printFold();
    }
    if (actualEditor->maxfold < fold)
    {
      actualEditor->maxfold = fold;
      printFold();
    }

    ZLINESPTR(actualEditor->wdy) = z;
    actualEditor->wdy =
          recalcTopOfWindow(actualEditor->wdy);
  }
}

/*  If cursor ontop line => zlinesptr[0]:  */
if (actualEditor->wdy == 0)
  ZLINESPTR(0) = z;

restoreZlinenptr();

/* redisplay window: */
actualEditor->xpos = autoindent;
if (actualEditor->xpos <= actualEditor->leftpos)
{
  /* If scrolling necessary: printAll */
  actualEditor->leftpos = actualEditor->xpos - 1;
  if (++actualEditor->wdy >= actualEditor->wch)
  {
    ZLINESPTR(0) = ZLINESPTR(1);
    ZLINESNR(0)  = ZLINESNR(1);
    restoreZlinenptr();
    actualEditor->wdy--;
  }
  actualEditor->ypos = ZLINESNR(actualEditor->wdy);

  printAll();
}
else
  if (1)
  {
    /* If fold marker: then redisplay window: */
    if (++actualEditor->wdy >= actualEditor->wch)

      ZLINESPTR(0) = ZLINESPTR(1);
      ZLINESNR(0)  = ZLINESNR(1);
      restoreZlinenptr();
      actualEditor->wdy--;
    }
    actualEditor->ypos = ZLINESNR(actualEditor->wdy);

    printAll();
  }
  else
  {
    /* it was scrolled: */
    if (++actualEditor->wdy >= actualEditor->wch)
    {
      /* Cursor was on last line: */
      register UWORD nr;
```

```
            scrollUp((UWORD) 1);
            nr = (--actualEditor->wdy) - 1;
            actualEditor->ypos = ZLINESNR(actualEditor->wdy);
            printLine(ZLINESPTR(nr),actualEditor->yoff
                        + actualEditor->ch*nr);
          }
          else
          {
            /* Cursor near middle of window: */
            register UWORD nr;

            if (ZLINESPTR((nr = actualEditor->wdy) + 1))
            {
              /* move rest of window down: */
              ScrollRaster(actualEditor->rp,
                      0L,(LONG)-actualEditor->ch,
                        (LONG)actualEditor->xoff,
                      (LONG)actualEditor->yoff
                      + actualEditor->wdy*actualEditor->ch,
                      (LONG)actualEditor->xscr,
                        (LONG)actualEditor->yscr);
            }
            actualEditor->ypos = ZLINESNR(actualEditor->wdy);

            printLine(ZLINESPTR(nr),actualEditor->yoff
                        + actualEditor->ch*nr);
            nr--;
            printLine(ZLINESPTR(nr),actualEditor->yoff
                        + actualEditor->ch*nr);
          }
        }

      printXpos();
      printYpos();
      printNumLines();
      printFlags();

      return (TRUE);
    }
    else
      return (FALSE);
  }
  else
  {
    cursorHome();
    cursorDown();

    return (FALSE);
  }
}
/*#ENDFD*/
/*#FOLD:  getLastWord */
/*************************************
 * getLastWord(str,len)
 * Returns pointer to last
 * word in a string.
 * str ^ String.
 * len = buffer length.
 *************************************/

UBYTE *getLastWord(str,len)
register UBYTE    *str;
```

```
register UWORD        len;
{
  register UBYTE c,*lwb,*lwn,*end;

  lwb = NULL;          /* Last Word is space  */
  lwn = NULL;          /* Last Word normal */
  end = str + len - 1;     /* pointer to string end */
  for (str += len; len; len--)
  {
    if ((c = *--str) == ' ')
    {
      lwb = str;
      break;
    }
    else if (! ((c >= 'A') && (c <= 'Z')
          || (c >= 'a') && (c <= 'z')
          || (c >= '0') && (c <= '9')
          || (c >= 192) || (c == '_')))
      if (lwn == NULL)
        lwn = str;
  }

  if (lwb == NULL)
    if (lwn)
      lwb = lwn;
    else
      lwb = end;
  else
    if (lwn)
      if ((end - lwb) > MAXWIDTH / 5)
        if ((end - lwn + 5) < (end - lwb))
          lwb = lwn;

  return (lwb + 1);
}
/*#ENDFD*/
/*#FOLD:  insertChar */
/*****************************************
 * insertChar(c)
 * Inserts character in buffer.
 * FALSE returned if character
 * cannot be inserted.
 * c= character.
 *****************************************/

BOOL insertChar(c)
register UBYTE  c;
{
  UBYTE *ptr;
  WORD inc,rest;
  WORD xadd = 1;

  if (! getBufferPointer(&ptr,&inc,&rest))
    return (FALSE);

  if ((c == TAB) && (actualEditor->tabs))
  {
    if (actualEditor->insert)
    {
      /* Insert spaces: */
      register UBYTE *p1,*p2;
      register UWORD num,l;
```

```
      p1  = actualEditor->tabstring + actualEditor->xpos - 1;
      l    = MAXWIDTH - actualEditor->xpos;
      num = 1;
      while (l && (*++p1))
      {
        xadd++;
        l--;
        num++;
      }

      /* num = Number of spaces to insert: */
      if (actualEditor->buflen + num > MAXWIDTH)
      {
        xadd -= num + actualEditor->buflen - MAXWIDTH;
        num = MAXWIDTH - actualEditor->buflen;
      }

      /* move rest of line: */
      p1 = ptr + rest;
      p2 = p1 + num;
      l   = rest;
      while (l)
      {
        *--p2 = *--p1;
        l--;
      }

      /* Insert spaces: */
      p1 = ptr;
      l   = num;
      while (l)
      {
        *p1 = ' ';
        p1++;
        l--;
      }

      actualEditor->buflen += num;
      rest += num;
    }
    else
    {
      /* move cursor only: */
      register UBYTE *tab;
      register UWORD max;

      tab = actualEditor->tabstring + actualEditor->xpos - 1;
      max = MAXWIDTH - actualEditor->xpos;
      while (max && (*++tab))
      {
        xadd++;
        max--;
      }
    }
  }
  else
  {
    /* besides a normal character: */
    if ((! actualEditor->skipblanks)
      &&  (actualEditor->xpos >= actualEditor->rm))
    {
```

```
      /* Word-Wrap when SkipBlanks == 0 */
      actualEditor->xpos = getLastWord(actualEditor->buffer,
                                       actualEditor->buflen)
                         - actualEditor->buffer + 1;
      if (! insertLine((UBYTE) 0))
        return (FALSE);

      if (! saveLine(actualEditor->actual))
        DisplayBeep(NULL);

      cursorEnd();

      if (! getBufferPointer(&ptr,&inc,&rest))
        return (FALSE);
    }

    if (actualEditor->insert)
      if (actualEditor->buflen < MAXWIDTH)
      {
        register UBYTE *p1,*p2;
        register UWORD l;

        p1 = ptr + rest;
        p2 = p1;
        l = rest;
        while (l)
        {
          *p2 = *--p1;
          p2--;
          l--;
        }
        *ptr = c;

        /* Lenght + 1 */
        actualEditor->buflen++;
        rest++;
      }
      else
        return (FALSE);
    else
    {
      *ptr = c;
      if (rest == 0)
      {
        actualEditor->buflen++;
        rest = 1;
      }
    }
  }

  /* move cursor: */
  while (xadd)
  {
    if (! cursorRight()) break;
    xadd--;
  }

  /* Line output: */
  printLine(actualEditor->actual,
    actualEditor->yoff + actualEditor->ch*actualEditor->wdy);

  return (TRUE);
```

```
}
/*#ENDFD*/
/*#FOLD:  deleteLine */
/*****************************
 * deleteLine:
 * delete the line,
 * that the cusor is on.
 * FALSE returned if
 * cannot be deleted
 *****************************/

BOOL deleteLine()
{
  register struct Zline *line,*next,*prev;
  register UWORD l = 0;
  UWORD nextnr,prevnr;

  if (line = ZLINESPTR(actualEditor->wdy))
  {
    /* erase reference to lien: */
    if (line == actualEditor->actual)
    {
      actualEditor->actual = NULL;
      actualEditor->bufypos = 0;
      actualEditor->buflen  = 0;
    }

    actualEditor->changed = 1;
    actualEditor->num_lines--;

    /* Reconstruct folding: */
    if (line->flags & ZLF_FSE)
    {
      register struct Zline *z;

      /* change fold level of following lines: */
      if ((z = line->succ)->succ)
      {
        if (((line->flags+1) & ZLF_FOLD) == (z->flags & ZLF_FOLD))
          /* line was fold start: */
          l = -1;
        else if ((line->flags & ZLF_FOLD) == ((z->flags+1) & ZLF_FOLD))
          /* line was fold end: */
          l = 1;
        else
          l = 0;

        if (l)
          while (z->succ)
          {
            z->flags = (z->flags & ~ZLF_FOLD)
                  | ((z->flags + l) & ZLF_FOLD);
            z = z->succ;
          }
      }
    }

    /* Note next and previous line: */
    prevnr= ZLINESNR(actualEditor->wdy);
    prev  = prevLine(line,&prevnr);
    next  = nextLine(line,&nextnr);
    nextnr= ZLINESNR(actualEditor->wdy);
```

```
/* Security check: */
while ((prev == NULL) && (next == NULL) && (actualEditor->minfold))
{
  l = 1;        /* Window completely output */
  actualEditor->minfold--;
  printFold();

  prevnr= ZLINESNR(actualEditor->wdy);
  prev  = prevLine(line,&prevnr);
  next  = nextLine(line,&nextnr);
  nextnr= ZLINESNR(actualEditor->wdy);

  if (prev || next) break;
  if (actualEditor->minfold) continue;

  if ((next = actualEditor->zlines.head)->succ == NULL)
    next = NULL;
  else
    nextnr = 1;

  break;
}

/* delete line: */
deleteZline(line);

/* pointer to actual line reset for recalc: */
if (next)
{
  ZLINESPTR(actualEditor->wdy) = next;
  ZLINESNR(actualEditor->wdy)  = nextnr;
}
else
{
  ZLINESPTR(actualEditor->wdy) = prev;
  ZLINESNR(actualEditor->wdy)  = prevnr;
}
/* Restore window contents: */
if (l)
{
  /* Redisplay window in each case: */
  actualEditor->wdy = recalcTopOfWindow(actualEditor->wdy);
  restoreZlinenptr();
  printAll();
  cursorOnText();
}
else
{
  /* with scrolling: */
  register UWORD oldwdy;
  oldwdy = actualEditor->wdy;
  actualEditor->wdy = recalcTopOfWindow(actualEditor->wdy);
  restoreZlinenptr();
  if (next)
  {
    /* push up rest of window: */
    ScrollRaster(actualEditor->rp,
                 0L, (LONG)actualEditor->ch,
                 (LONG)actualEditor->xoff,
                 (LONG)actualEditor->yoff
                 + actualEditor->wdy*actualEditor->ch
```

```
                    (LONG)actualEditor->xscr,
                    (LONG)actualEditor->yscr);
          printLine(ZLINESPTR(actualEditor->wch - 1),
                actualEditor->yoff
                + actualEditor->ch*(actualEditor->wch - 1));
        }
        else
          if (oldwdy == actualEditor->wdy)
          {
            /* scroll top section of window down: */
            ScrollRaster(actualEditor->rp,
                      0L,(LONG)-actualEditor->ch,
                      (LONG)actualEditor->xoff,
                      (LONG)actualEditor->yoff,
                      (LONG)actualEditor->xscr,
                      (LONG)actualEditor->yoff
                      + (actualEditor->wdy + 1)
                       *actualEditor->ch - 1);
            printLine(ZLINESPTR(0),actualEditor->yoff);
          }
          else
            printLine(ZLINESPTR(oldwdy),actualEditor->yoff
                  + actualEditor->ch*oldwdy);
        cursorOnText();
      }
      printXpos();
      printYpos();
      printNumLines();
      printFlags();
      return (TRUE);
    }
    else
      return (FALSE);
}
/*#ENDFD*/
/*#FOLD:  initFolding */
/*********************************
 * initFolding:
 * calculate the fold level of all
 * lines from text start.
 *********************************/
void initFolding()
{
  register struct Zline *z;
  register UWORD level,l;
  z = actualEditor->zlines.head;
  level = 0;
  while (z->succ)
  {
    z->flags = (z->flags & ~ZLF_FOLD) | (level & ZLF_FOLD);
    if (l = getFoldInc(z))
    {
      z->flags |= ZLF_FSE;
      level += l;
    }
    z = z->succ;
  }
}
/*#ENDFD*/
```

Version 6 Cursor.c

```
/*************************
 *
 * Module: Cursor.c
 *
 * Moving the cursors.
 *
 *************************/

/*#FOLD: Includes */
/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"
/*#ENDFD*/
/*#FOLD: Defines */
/***********
 *
 * Defines:
 *
 ***********/

#define CSI 0x9B
#define CUU 'A'
#define CUD 'B'
#define CUF 'C'
#define CUB 'D'
#define SU 'S'
#define SD 'T'
#define CHOME 'A'
#define CEND '@'

#define ESC 27
#define UNDO '?'
#define CFOLD 6
#define CENDF 5
#define DELLINE 2
#define DEL 127
#define BS 8
#define TAB 9
#define LF 10
#define CR 13
#define REPCOM 7
/*#ENDFD*/
/*#FOLD: External Functions */
/**********************
 *
 * External Functions:
 *
 **********************/

void SetDrMd(),RectFill(),printAll(),DisplayBeep(),undoLine();
```

```
void ScrollRaster(),cursorOnText(),printXpos(),printYpos();
void printFold(),printFlags(),cursorOnText();
struct Zline *nextLine(),*prevLine();
BOOL insertChar(),deleteChar(),backspaceChar(),insertLine(),deleteLine();
BOOL ActivateGadget();
UBYTE *executeCommand();
/*#ENDFD*/
/*#FOLD: External Variables */
/*********************
 *
 * External Variables:
 *
 *********************/

extern struct Editor *actualEditor;
extern UWORD SplitDec;
extern struct Gadget G_Command;
/*#ENDFD*/

/***************
 *     *
 * Functions: *
 *     *
 ***************/

/*#FOLD: restoreZlinenptr */
/***********************************
 *
 * restoreZlinenptr:
 *
 * Restore the zlinesptr/nr field
 * for all lines in window plus
 * the next line.
 * The first line must be
 * correct!
 *
 ***********************************/

void restoreZlinenptr()
{
  register UWORD n;
  register struct Zline **zptr,*z;
  register UWORD *pnr;
  UWORD znr;

  for (n = actualEditor->wch, zptr = actualEditor->zlinesptr,
          pnr = actualEditor->zlinesnr,
          z = *zptr++, znr = *pnr++; n; n--)
  {
    *zptr++ = (z = nextLine(z,&znr));
    *pnr++  = znr;
  }
}
/*#ENDFD*/
/*#FOLD: Cursor */
/*********************************
 *
 * Cursor:
 *
 * Set or erase the cursor
 * at the actual position.
 *
```

```
 *********************************/

void Cursor()
{
  register struct RastPort *rp;
  register LONG x,y;
  register LONG cw,ch;

  SetDrMd(rp = actualEditor->rp,COMPLEMENT);

  x = (actualEditor->xpos - actualEditor->leftpos - 1)
    *(cw = actualEditor->cw) + actualEditor->xoff;
  y = actualEditor->wdy
    *(ch = actualEditor->ch) + actualEditor->yoff;

  RectFill(rp,x,y,x + cw-1,y + ch-1);

  SetDrMd(rp,JAM2);
}
/*#ENDFD*/
/*#FOLD: scrollRight */
/*******************************
 *
 * scrollRight(num):
 *
 * Scroll the window contents
 * num characters to the right.
 *
 * num = number of characters.
 *
 *******************************/

void scrollRight(num)
register UWORD    num;
{
  register UWORD y,ch,count;
  register struct Zline **z;
  UWORD wcw,xoff,leftpos;

  actualEditor->leftpos -= num;

  if (num >= actualEditor->wcw)
    printAll();
  else
  {
    /* Scrolling: */
    ScrollRaster(actualEditor->rp,
             -num * (LONG)actualEditor->cw,0L,
             (LONG)actualEditor->xoff,(LONG)actualEditor->yoff,
             (LONG)actualEditor->xscr,(LONG)actualEditor->yscr);

    /* now list printAll(): */
    wcw = actualEditor->wcw;
    actualEditor->wcw = num + 1;
    SplitDec = 1;

    y   = actualEditor->yoff;
    ch    = actualEditor->ch;
    count = actualEditor->wch;

    /* Output line: */
    for (z = actualEditor->zlinesptr; count-- && (*z != NULL);
```

```
        z++, y += ch)
      printLine(*z,y);

    SplitDec = 0;

    /* Display right border beside last character */
    leftpos = actualEditor->leftpos;
    xoff    = actualEditor->xoff;
    actualEditor->wcw     = 2;
    actualEditor->leftpos += (y = wcw - 2);
    actualEditor->xoff    += actualEditor->cw * y;

    y     = actualEditor->yoff;
    count = actualEditor->wch;

    /* Output line: */
    for (z = actualEditor->zlinesptr; count-- && (*z != NULL);
        z++, y += ch)
      printLine(*z,y);

    actualEditor->wcw    = wcw,
    actualEditor->leftpos= leftpos;
    actualEditor->xoff   = xoff;
  }
}
/*#ENDFD*/
/*#FOLD: scrollLeft */
/*******************************
 *
 * scrollLeft(num):
 *
 * Scroll the window contents
 * num characters left.
 *
 * num = number of characters.
 *
 *******************************/

void scrollLeft(num)
register UWORD  num;
{
  register UWORD y,ch,count;
  register struct Zline **z;
  UWORD wcw,xoff,leftpos;

  actualEditor->leftpos += num;

  if (num >= actualEditor->wcw)
    printAll();
  else
  {
    /* Scrolling: */
    ScrollRaster(actualEditor->rp,
             num * (LONG)actualEditor->cw,0L,
             (LONG)actualEditor->xoff,(LONG)actualEditor->yoff,
             (LONG)actualEditor->xscr,(LONG)actualEditor->yscr);

    /* now list printAll(): */
    wcw     = actualEditor->wcw;
    leftpos = actualEditor->leftpos;
    xoff    = actualEditor->xoff;
    actualEditor->wcw     = num + 1;
```

```
    actualEditor->leftpos += (y = wcw - num - 1);
    actualEditor->xoff    += actualEditor->cw * y;

    y     = actualEditor->yoff;
    ch    = actualEditor->ch;
    count = actualEditor->wch;

    /* Output line: */
    for (z = actualEditor->zlinesptr; count-- && (*z != NULL);
        z++, y += ch)
      printLine(*z,y);

    actualEditor->wcw    = wcw;
    actualEditor->leftpos= leftpos;
    actualEditor->xoff   = xoff;
  }
}
/*#ENDFD*/
/*#FOLD: scrollUp */
/*******************************
 *
 * scrollUp(num):
 *
 * Scroll the window contents
 * num lines up.
 * Num lines must
 * exist!
 *
 * num = Number of the line.
 *
 *******************************/

void scrollUp (num)
register UWORD num;
{
  register struct Zline *z,**zptr,**zold;
  register UWORD n,wch = actualEditor->wch,y;
  UWORD znr,*pnr,*oldnr;

  if (num >= wch)
  {
    /* Display window without Scrolling */
    /* First look for top line: */
    n = wch;
    if (num > wch)
    {
      z = ZLINESPTR(n);
      znr = ZLINESNR(n);
      while (++n < num)
        z = nextLine(z,&znr);
    }
    else
    {
      z = ZLINESPTR(n - 1);
      znr = ZLINESNR(n - 1);
    }

    /* z points before the top line of the window */
    for (n = 0, zptr = actualEditor->zlinesptr,
        pnr = actualEditor->zlinesnr; n <= wch; n++)
    {
      *zptr++ = (z = nextLine(z,&znr));
```

```
        *pnr++  = znr;
      }

      /* Now redisplay the window: */
      printAll();
    }
    else
    {
      /* Scrolling: */
      ScrollRaster(actualEditor->rp,
               0L,num * (LONG)actualEditor->ch,
               (LONG)actualEditor->xoff, (LONG)actualEditor->yoff,
               (LONG)actualEditor->xscr, (LONG)actualEditor->yscr);

      /* zlinesptr/nr recalculated: */
      zptr = actualEditor->zlinesptr;
      zold = &(ZLINESPTR(num));
      pnr = actualEditor->zlinesnr;
      oldnr = &(ZLINESNR(num));
      for (n = 0; n <= wch - num; n++)
      {
        *zptr++ = *zold++;
        *pnr++  = *oldnr++;
      }

      z = *--zold;
      zold = zptr - 1;        /* mark for output! */
      znr = *--oldnr;
      oldnr = pnr - 1;
      while (n <= wch)
      {
        *zptr++ = (z = nextLine(z,&znr));
        *pnr++ = znr;
        n++;
      }

      /* Output new line: */
      for (n = num, y = actualEditor->yoff+(wch-num)*actualEditor->ch;
          (n && (*zold != NULL)); n--, zold++, y += actualEditor->ch)
        printLine(*zold,y);
    }
}
/*#ENDFD*/
/*#FOLD: scrollDown */
/*******************************
 *
 * scrollDown(num):
 *
 * Scroll the window contents
 * num lines down.
 * Num lines must
 * exist!
 *
 * num = number of lines.
 *
 *******************************/

void scrollDown (num)
register UWORD num;
{
  register struct Zline *z,**zptr,**zold;
  register UWORD n,wch = actualEditor->wch,y;
```

```
    UWORD znr,*pnr,*oldnr;

    if (num >= wch)
    {
      /* Display window without Scrolling */
      /* search backwards fo rtop line: */
      for (z = ZLINESPTR(0), znr = ZLINESNR(0), n = num - wch; n; n--)
        z = prevLine(z,&znr);

      /* z points to first line after bottom window edge. */
      for (n = 0, zptr = &(ZLINESPTR(wch)), pnr = &(ZLINESNR(wch));
          n <= wch; n++)
      {
        *zptr-- = z;
        *pnr-- = znr;
        z = prevLine(z,&znr);
      }

      /* Now redisplay the window: */
      printAll();
    }
    else
    {
      /* Scrolling: */
      ScrollRaster(actualEditor->rp,
               0L,-num * (LONG)actualEditor->ch,
               (LONG)actualEditor->xoff,(LONG)actualEditor->yoff,
               (LONG)actualEditor->xscr,(LONG)actualEditor->yscr);

      /* zlinesptr recalculated: */
      zptr = &(ZLINESPTR(wch));
      zold = &(ZLINESPTR(wch - num));
      pnr = &(ZLINESNR(wch));
      oldnr = &(ZLINESNR(wch - num));
      for (n = 0; n <= wch - num; n++)
      {
        *zptr-- = *zold--;
        *pnr--  = *oldnr--;
      }

      for (z = *++zold, znr = *++oldnr, n = num; n; n--)
      {
        *zptr-- = (z = prevLine(z,&znr));
        *pnr--  = znr;
      }

      /* Output new line: */
      for (n = num, y = actualEditor->yoff; (n && (*zold != NULL));
          n--, zold++, y += actualEditor->ch)
        printLine(*zold,y);
    }
}
/*#ENDFD*/
/*#FOLD: cursorLeft */
/*****************************************
 *
 * cursorLeft:
 *
 * moce cursor one position
 * left, False returned if cursor,
 * is in first column.
 *
```

```
 ****************************************/

BOOL cursorLeft()
{
  if (actualEditor->xpos > 1)
  {
    actualEditor->xpos--;
    if (actualEditor->xpos <= actualEditor->leftpos)
      scrollRight((UWORD)1);

    printXpos();
    return (TRUE);
  }
  else
    return (FALSE);
}
/*#ENDFD*/
/*#FOLD: cursorRight */
/****************************************
 *
 * cursorRight:
 *
 * moce cursor one position
 * right, False returned if cursor,
 * is in last column.
 *
 ****************************************/

BOOL cursorRight()
{
  if (actualEditor->xpos < MAXWIDTH)
  {
    actualEditor->xpos++;
    if (actualEditor->xpos
      >= (actualEditor->leftpos + actualEditor->wcw))
      scrollLeft((UWORD)1);

    printXpos();
    return (TRUE);
  }
  else
    return (FALSE);
}
/*#ENDFD*/
/*#FOLD: cursorUp */
/****************************************
 *
 * cursorUp:
 *
 * Move cursor one line up.
 * False returned when
 * on first line.
 *
 ****************************************/

BOOL cursorUp()
{
  UWORD help;

  if (actualEditor->wdy)
    actualEditor->wdy--;
  else
```

```
    if (prevLine(ZLINESPTR(0),&help))
      scrollDown((UWORD)1);
    else
      return (FALSE);

  actualEditor->ypos = ZLINESNR(actualEditor->wdy);
  printYpos();

  return (TRUE);
}
/*#ENDFD*/
/*#FOLD: cursorDown */
/****************************************
 *
 * cursorDown:
 *
 * Move cursor one line down.
 * False returned when
 * on last line.
 *
 ****************************************/

BOOL cursorDown()
{
  if (ZLINESPTR(actualEditor->wdy + 1))
  {
    if (++actualEditor->wdy >= actualEditor->wch)
    {
      scrollUp((UWORD)1);
      actualEditor->wdy--;
    }
    actualEditor->ypos = ZLINESNR(actualEditor->wdy);
    printYpos();

    return (TRUE);
  }
  else
    return (FALSE);
}
/*#ENDFD*/
/*#FOLD: halfPageUp */
/*********************************
 *
 * halfPageUp:
 *
 * Move cursor on half page
 * up. The text is scrolled
 * half page down
 *
 * False returned when
 * in first line.
 *
 *********************************/

BOOL halfPageUp()
{
  register UWORD max,n;
  register struct Zline *z;
  UWORD help;

  max = actualEditor->wch / 2;
  for (n = 0, z = ZLINESPTR(0); n < max; n++)
```

```
    if (!(z = prevLine(z,&help)))
      break;

  if (n)
  {
    scrollDown(n);
    actualEditor->ypos = ZLINESNR(actualEditor->wdy);
    printYpos();

    return (TRUE);
  }
  else
    return (FALSE);
}
/*#ENDFD*/
/*#FOLD: halfPageDown */
/*********************************
 *
 * halfPageDown:
 *
 * Move cursor on half page
 * down. The text is scrolled
 * half page down.
 *
 * False returned when
 * on last line.
 *
 *********************************/

BOOL halfPageDown()
{
  register UWORD max,n;
  register struct Zline *z;
  UWORD help;

  max = actualEditor->wch / 2;
  for (n = 0, z = ZLINESPTR(actualEditor->wch - 1); n < max; n++)
    if (!(z = nextLine(z,&help)))
      break;

  if (n)
  {
    scrollUp(n);
    actualEditor->ypos = ZLINESNR(actualEditor->wdy);
    printYpos();

    return (TRUE);
  }
  else
    return (FALSE);
}
/*#ENDFD*/
/*#FOLD: cursorHome */
/*********************************
 *
 * cursorHome:
 *
 * move cursor to first line.
 *
 *********************************/

BOOL cursorHome()
```

```
{
  actualEditor->xpos = 1;
  if (actualEditor->leftpos)
    scrollRight(actualEditor->leftpos);

  printXpos();

  return (TRUE);
}
/*#ENDFD*/
/*#FOLD: cursorEnd */
/*************************************
 *
 * cursorEnd:
 *
 * Set the cursor on the last line.
 *
 *************************************/

BOOL cursorEnd()
{
  register UBYTE *ptr,*tab;
  register UWORD len,pos;
  register struct Zline *z;

  if (actualEditor->bufypos)
    pos = actualEditor->buflen + 1;
  else
    if (z = ZLINESPTR(actualEditor->wdy))
    {
      /* remove CR from lenght: */
      len = z->len;
      ptr = ((UBYTE *)(z + 1)) + len - 1;
      if (len)
        if (*ptr == CR)
          len--;
        else if (*ptr == LF)
          if (--len)
            if (*--ptr == CR)
              len--;

      if (actualEditor->tabs)
      {
        ptr = ((UBYTE *)(z + 1));
        tab = actualEditor->tabstring;
        pos = 1;

        while ((len--) && (pos < MAXWIDTH))
          if (*ptr++ == TAB)
            do
            {
              tab++;
              pos++;
            } while ((*tab) && (pos < MAXWIDTH));
          else
          {
            tab++;
            pos++;
          }
      }
      else
        pos = len + 1;
```

```
    }
    else
      return (FALSE);

  actualEditor->xpos = pos;
  if (pos >= (actualEditor->leftpos + actualEditor->wcw))
    scrollLeft(pos + 1 - (actualEditor->leftpos
                  + actualEditor->wcw));
  else if (pos <= actualEditor->leftpos)
    scrollRight(actualEditor->leftpos + 1 - pos);

  printXpos();

  return (TRUE);
}
/*#ENDFD*/
/*#FOLD: recalcTopOfWindow */
/***********************************
 *
 * recalcTopOfWindow(y)
 *
 * Calculate the cursor position
 * in line y of
 * line in window.
 * Get new Y-Position.
 *
 * y = Number of line [0,wdy[
 *
 ***********************************/

UWORD recalcTopOfWindow(y)
register UWORD        y;
{
  register struct Zline *z,*zp;
  register UWORD znrp;
  UWORD znr,oldy = y;

  z    = ZLINESPTR(y);
  znr = ZLINESNR(y);
  while (y)
  {
    zp    = z;
    znrp  = znr;
    if ((z = prevLine(z,&znr)) == NULL)
      break;
    y--;
  }

  if (y)
  {
    ZLINESPTR(0) = zp;
    ZLINESNR(0)  = znrp;
  }
  else
  {
    ZLINESPTR(0) = z;
    ZLINESNR(0)  = znr;
  }

  return (oldy - y);
}
/*#ENDFD*/
```

```
/*#FOLD: cursorTop */
/*****************************
 *
 * cursorTop:
 *
 * Set Cursor to to of  Text
 * first line.
 *
 *****************************/

BOOL cursorTop()
{
  register struct Zline *z;
  UWORD n;
  register UWORD cnt;

  z  = ZLINESPTR(0);
  n  = ZLINESNR(0);
  cnt = 0;
  while (z = prevLine(z,&n))
    cnt++;

  if (cnt)
    scrollDown(cnt);
  actualEditor->ypos = ZLINESNR(actualEditor->wdy = 0);
  printYpos();

  return (TRUE);
}
/*#ENDFD*/
/*#FOLD: cursorBottom */
/*****************************
 *
 * cursorBottom:
 *
 * Set Cursor to bottom of text
 * last line.
 *
 *****************************/

BOOL cursorBottom()
{
  register struct Zline *z;
  UWORD n;
  register UWORD wch,cnt;

  if   (z  = ZLINESPTR(wch = actualEditor->wch   - 1))
  {
    n  =  ZLINESNR(wch);
    cnt = 0;
    while (z = nextLine(z,&n))
      cnt++;

    if (cnt)
      scrollUp(cnt);

    actualEditor->ypos =  ZLINESNR(actualEditor->wdy    = wch);
  }
  else
  {
    actualEditor->ypos =  ZLINESNR(actualEditor->wdy    = wch);
    cursorOnText();
```

```
  }

  printYpos();

  return (TRUE);
}
/*#ENDFD*/
/*#FOLD: enterFold */
/***************************
 *
 * enterFold:
 *
 * Step one fold in.
 *
 ***************************/

void enterFold()
{
  register struct Zline *z;

  if (actualEditor->maxfold < ZLF_FOLD)
  {
    actualEditor->maxfold++;
    if (z = ZLINESPTR(actualEditor->wdy))
      if ((z = z->succ)->succ)
        /* If there is a following line: */
        if (((ZLINESPTR(actualEditor->wdy)->flags & ZLF_FOLD)
              < (z->flags & ZLF_FOLD))
          && ((z->flags & ZLF_FOLD) == actualEditor->maxfold))
        {
          /* When this begins a new fold: */
          actualEditor->minfold = actualEditor->maxfold;
          ZLINESPTR(0)   = z;
          ZLINESNR(0)    = ZLINESNR(actualEditor->wdy) + 1;
          actualEditor->wdy = 0;
        }

    actualEditor->wdy = recalcTopOfWindow(actualEditor->wdy);
    restoreZlinenptr();
    printAll();
    actualEditor->ypos = ZLINESNR(actualEditor->wdy);
    printYpos();
    printFold();
  }
}
/*#ENDFD*/
/*#FOLD: exitFold */
/***************************
 *
 * exitFold:
 *
 * Step one fold out.
 *
 ***************************/

void exitFold()
{
  register UWORD wdy;

  if (actualEditor->maxfold)
  {
    actualEditor->maxfold--;
```

```
    if (actualEditor->minfold > actualEditor->maxfold)
      actualEditor->minfold = actualEditor->maxfold;

    wdy = actualEditor->wdy;
    if (ZLINESPTR(wdy) == NULL)
      actualEditor->wdy = wdy = 0;

    if (ZLINESPTR(wdy))
      if (((ZLINESPTR(wdy)->flags & ZLF_FOLD) > actualEditor->maxfold)
      || ((ZLINESPTR(wdy)->flags & ZLF_FOLD) < actualEditor->minfold))
        ZLINESPTR(wdy) = prevLine(ZLINESPTR(wdy),&(ZLINESNR(wdy)));

    actualEditor->wdy = recalcTopOfWindow(wdy);
    restoreZlinenptr();
    printAll();
    cursorOnText();
    printYpos();
    printFold();
  }
}
/*#ENDFD*/
/*#FOLD: handleKeys */
/*******************************
 *
 * handleKeys(buf,len):
 *
 * Handles keyboard messages.
 * Give FALSE, if
 * the Editor commands
 * elset TRUE.
 *
 * buf ^ buffer with character.
 * len = number of characters.
 *
 *******************************/

BOOL handleKeys(buf,len)
register UBYTE  *buf;
register WORD    len;
{
  register UBYTE first;

  while (len > 0)
  {
    first = *buf;
    buf++;
    len--;
    if ((first == CSI) && (len > 0))
    {
      len--;
      switch (*buf++)
      {
        case CUU:
          cursorUp();
          break;
        case CUD:
          cursorDown();
          break;
        case CUF:
          cursorRight();
          break;
        case CUB:
```

```
            cursorLeft();
            break;
          case SU:
            halfPageDown();
            break;
          case SD:
            halfPageUp();
            break;
          case ' ':
            if (len > 0)
            {
              len--;
              switch (*buf++)
              {
                case CHOME:
                  cursorHome();
                  break;
                case CEND:
                  cursorEnd();
                  break;
                default:
                  if (! insertChar((UBYTE)CSI))
                    DisplayBeep(NULL);
                  buf -= 2;
                  len += 2;
              }
              break;
            }
            else
              goto noCSI;
          case UNDO:
            if ((len > 0) && (*buf == '~'))
            {
              buf++;
              len--;
              undoLine();
              break;
            }
            /* Else: default */
          default:
noCSI:
            /* No command-Sequence! */
            if (! insertChar((UBYTE)CSI))
              DisplayBeep(NULL);

            buf--;
            len++;
            break;
        } /* switch (first) */
      }
      else
        switch (first)
        {
          case ESC:
            /* Rest of buf transferred: */
            {
              register UBYTE *sibuf;
              register UWORD cnt;

              sibuf = actualEditor->gc_SIBuffer;
              while (len)
              {
```

```
        *sibuf++ = *buf++;
        len--;
      }
      *sibuf = 0;

      cnt = (sibuf - actualEditor->gc_SIBuffer);
      actualEditor->gc_GadgetSI.BufferPos  = cnt;
      actualEditor->gc_GadgetSI.NumChars   = cnt;
      actualEditor->gc_GadgetSI.DispPos    = 0;
    }

    /* Gagdet aktivated: */
    if (!ActivateGadget(&(actualEditor->G_Command),
                  actualEditor->window,NULL))
      DisplayBeep(NULL);
    break;
  case REPCOM:
  {
    register UBYTE *ptr;
    register SHORT pos,hw;

    if (ptr = executeCommand(actualEditor->gc_SIBuffer))
      if (ptr == -1L)
        return (FALSE);
      else
      {
        if   (ptr >= 0x80000000)
          ptr = NULL - ptr;

        /* Cursor in StringGadget in error set */
        pos = ptr - actualEditor->gc_SIBuffer;
        actualEditor->gc_GadgetSI.BufferPos = pos;
        if (pos < actualEditor->gc_GadgetSI.DispCount)
          actualEditor->gc_GadgetSI.DispPos = 0;
        else
          actualEditor->gc_GadgetSI.DispPos =  os
              - actualEditor->gc_GadgetSI DispCount / 2;

        /* Gagdet aktivated: */
        ActivateGadget(&(actualEditor->G_Command),
                    actualEditor->window,NULL);
        DisplayBeep(NULL);

        /* return now: */
        return (TRUE);
      }

    break;
  }
  case CFOLD:
    enterFold();
    break;
  case CENDF:
    exitFold();
    break;
  case DELLINE:
    if (! deleteLine())
      DisplayBeep(NULL);
    break;
  case DEL:
    if (! deleteChar())
      DisplayBeep(NULL);
```

```
            break;
          case BS:
            if (! backspaceChar())
              DisplayBeep(NULL);
            break;
          case LF:
          case CR:
            if (! insertLine((UBYTE) LF))
              DisplayBeep(NULL);
            break;
          default:
            if (! insertChar(first))
              DisplayBeep(NULL);

        }   /* switch (first) */
      }     /* while (len) */

      return (TRUE);
}
/*#ENDFD*/
```

Version 6 Output.c

```
/***********************************
 *
 * Module: Output
 *
 * Output text in a window.
 *
 ***********************************/

/*#FOLD: Includes */
/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include "src/Editor.h"
/*#ENDFD*/
/*#FOLD: Defines */
/***********
 *
 * Defines:
 *
 ***********/

#define CR 13
#define LF 10
#define TAB 9
/*#ENDFD*/
/*#FOLD: External Functions */
/**********************
 *
 * External Functions:
 *
 **********************/

void SetAPen(),RectFill(),Text(),Move(),SetFont();
struct Zline *nextLine();
/*#ENDFD*/
/*#FOLD: External Variables */
/*********************
 *
 * External Variables:
 *
 *********************/

extern struct Editor *actualEditor;
extern struct TextFont *infoFont;
/*#ENDFD*/
/*#FOLD: Global Variables */
/*********************
 *
 * Global Variables:
 *
 *********************/
```

```
UWORD SplitDec = 0;
/*#ENDFD*/

/***************
 *             *
 * Functions: *
 *             *
 ***************/

/*#FOLD: initWindowSize */
/*********************************
 *
 * initWindowSize(ed)
 *
 * Initializes the width/height
 * statement of the actual windows.
 * Restores the zlineptr field
 *
 *
 * ed ^ Editor structure.
 *
 *********************************/

void initWindowSize    (ed)
register struct Editor *ed;
{
  register struct Window *w;
  register struct RastPort *rp;
  register UWORD n,newh;

  w   = ed->window;
  rp = ed->rp;

  ed->xoff = (UWORD)w->BorderLeft;
  ed->yoff = (UWORD)w->BorderTop;

  ed->cw   = (rp->TxWidth)? rp->TxWidth : 8;
  ed->ch   = (rp->TxHeight)? rp->TxHeight : 8;

  ed->wcw = (w->Width - ed->xoff - w->BorderRight) / ed->cw;
  newh    = (w->Height- ed->yoff - w->BorderBottom)/ ed->ch;

  ed->xrl = ed->xoff + ed->wcw*ed->cw;
  ed->xrr = w->Width - w->BorderRight - 1;
  ed->yru = w->Height - w->BorderBottom - 1;
  ed->bl  = rp->TxBaseline;

  {
    /* Position for Infoline: */
    register ULONG xinc,xsize;

    xinc = w->Width + ed->G_Info.LeftEdge + ed->gi_IText.LeftEdge - 1;
    xsize= infoFont->tf_XSize;

    ed->ip_xpos      = xinc + INFO_XPOS_INC * xsize;
    ed->ip_ypos      = xinc + INFO_YPOS_INC * xsize;
    ed->ip_numzlines  = xinc + INFO_NUMOFLINES_INC * xsize;
    ed->ip_minfold    = xinc + INFO_MINFOLD_INC * xsize;
    ed->ip_maxfold    = xinc + INFO_MAXFOLD_INC * xsize;
    ed->ip_flags     = xinc + INFO_FLAGS_INC * xsize;
    ed->ip_topedge    = w->Height + ed->G_Info.TopEdge
```

```
                + ed->gi_IText.TopEdge
                + (ULONG)infoFont->tf_Baseline - 1;
    }

    /* Now restore the zlineptr field: */
    if (newh != ed->wch)
    {
      register struct Zline *z,**zptr;
      UWORD znr,*pnr;

      if (newh < ed->wch)
      {
        /* Pointer to Null set: */
        zptr = ed->zlinesptr + newh + 1;
        for (n = newh; n < ed->wch; n++)
          *zptr++ = NULL;
      }
      else
      {
        zptr  = ed->zlinesptr + ed->wch;
        z     = *zptr++;
        pnr = &(ed->zlinesnr[ed->wch]);
        znr   = *pnr++;
        for (n = ed->wch; n < newh; n++)
        {
          *zptr++ = (z = nextLine(z,&znr));
          *pnr++  = znr;
        }
      }

      ed->wch = newh;
    }

    /* Top/bottom border for scrolling: */
    ed->xscr = ed->xoff + ed->wcw*ed->cw - 1;
    ed->yscr = ed->yoff + ed->wch*ed->ch - 1;
}
/*#ENDFD*/
/*#FOLD: convertLineForPrint */
/*****************************************
 *
 * convertLineForPrint(line,len,w,buf,lc)
 *
 * Convert line so that this can be
 * displayed simply.
 * Tabs converted into spaces
 *
 *
 * line ^ Zline.
 * len = the length.
 * w = maximum wiidth.
 * buf ^buffer, in which the converted line
 *       is stored.
 * lc ^ to Variable.
 *
 *****************************************/

UWORD convertLineForPrint(line,len,w,buf,lc)
UBYTE               *line;
register WORD             len;
register UWORD                w;
register UBYTE                 *buf;
```

```
UBYTE                           *lc;
{
  register UBYTE *tab;
  register UWORD l = 1, skip = actualEditor->leftpos;
  UBYTE lastchar;
  UWORD realLen;

  /* determine if ends with CR or LF: */
  if (len)
  {
    tab = line + len - 1;
    if (*tab == CR)
    {
      len--;
      lastchar = ' ';
    }
    else if (*tab == LF)
    {
      len--;
      lastchar = ' ';
      if ((len) && (*--tab == CR))
        len--;
    }
    else
      lastchar = 183;
  }
  else
    lastchar = 183;

  if (actualEditor->tabs)
  {
    /* line convert: */
    tab = actualEditor->tabstring;
    while ((len--) && (l < w))
      if ((*buf = *line++) == TAB)
        do
        {
          tab++;
          if (skip)
            skip--;
          else
          {
            l++;
            *buf++ = ' ';
          }
        } while ((*tab) && (l < w));
      else
      {
        tab++;
        if (skip)
          skip--;
        else
        {
          l++;
          buf++;
        }
      }
  }
  else
  {
    /* line copied with no tabs: */
    while ((len--) && (l < w))
```

```
    {
      if (skip)
        skip--;
      else
      {
        l++;
        *buf++ = *line;
      }
      line++;
    }
  }

  /* determine if the line is not completely converted: */
  if (len >= 0)
    lastchar = 183;

  /* save lenght of line */
  realLen = l - 1;

  /* line filled from last character to end: */
  while (l++ <= w)
    *buf++ = lastchar;

  *lc = lastchar;
  return (realLen);
}
/*#ENDFD*/
/*#FOLD: printAt */
/************************************
 *
 * printAt(buf,len,x,y,fgpen)
 *
 *Prive character string buf a position
 * (x,y) in a window. The character-
 * string is changed!
 *
 * buf ^ character sring.
 * len = the length.
 * x = Pixel-X-Position.
 * y = Pixel-Y-Position.
 * fgpen = Foreground color (Folding)
 *
 ************************************/

void printAt   (buf,len,x,y,fgpen)
register UBYTE  *buf;
WORD            len;
register UWORD        x,y;
ULONG                fgpen;
{
  struct RastPort *rp = actualEditor->rp;
  register UWORD cw = actualEditor->cw;
  register UWORD x2;
  UWORD y2 = y + actualEditor->ch - 1;
  UBYTE *ptr;
  ULONG l;

  while (len > 0)
    if (*buf == ' ')
    {
      /* output spaces: */
      x2 = x - 1;
```

```
        while ((*buf == ' ') && len--)
        {
          buf++;
          x2 += cw;
        }

        SetAPen(rp,BGPEN);
        RectFill(rp,(ULONG)x,(ULONG)y,(ULONG)x2,(ULONG)y2);
        SetAPen(rp,fgpen);

        x = ++x2;
      }
      else
      {
        /* Also output text: */
        Move(rp,(ULONG)x,(ULONG)y + actualEditor->bl);
        ptr = buf;

        if (CONTROLCODE(*buf))
        {
          SetAPen(rp,CTRLPEN);

          while (CONTROLCODE(*buf) && len--)
            *buf++ |= 64;
          l = buf - ptr;
          Text(rp,ptr,l);

          SetAPen(rp,fgpen);
        }
        else
        {
          while (!CONTROLCODE(*buf) && (*buf != ' ') && len--)
            buf++;
          l = buf - ptr;
          Text(rp,ptr,l);
        }
        x += cw*l;
      }
}
/*#ENDFD*/
/*#FOLD: printLine */
/************************************
 *
 * printLine(line,y)
 *
 * Displays Zline at pixel position
 *y on the screen.
 *
 * line ^ Zline structure.
 * y = pixel positon.
 *
 ************************************/

void printLine        (line,y)
register struct Zline *line;
register UWORD             y;
{
  static UBYTE buf[MAXWIDTH];

  if (line)
  {
```

```
register struct Zline *z;
register UBYTE *p1,*p2;
register UWORD w;
register ULONG pen;
UBYTE lc;

/* Test, if line even edited: */
if (line->flags & ZLF_USED)
{
  /* Buffer copies to  buf,since  printAt changed this! */
  if (MAXWIDTH > actualEditor->leftpos)
  {
    w = MAXWIDTH - actualEditor->leftpos;
    p2 = actualEditor->buffer + actualEditor->leftpos;

    p1 = buf;
    while (w)
    {
      *p1 = *p2;
      p1++;
      p2++;
      w--;
    }
  }
  else
    p1 = buf;

  /* fill rest from lastchar : */
  p2 = buf + (w = actualEditor->wcw);
  lc = actualEditor->buflastchar;
  while (p1 < p2)
  {
    *p1 = lc;
    p1++;
  }
}
else
  convertLineForPrint(line+1,line->len,
                w = actualEditor->wcw,buf,&lc);

/* If start/end marker of fold => FOLDPEN */
if (line->flags & ZLF_FSE)
{
  if ((z = line->succ)->succ)
  {
    if ((z->flags & ZLF_FOLD) <= actualEditor->maxfold)
    {
      SetAPen(actualEditor->rp,FOLDPEN);
      pen = FOLDPEN;
    }
    else
      pen = FGPEN;
  }
  else
  {
    SetAPen(actualEditor->rp,FOLDPEN);
    pen = FOLDPEN;
  }

  printAt(buf,w - SplitDec,actualEditor->xoff,y,pen);

  /* Old write color first again: */
```

```
        if (pen != FGPEN)
          SetAPen(actualEditor->rp,FGPEN);
      }
      else
        printAt(buf,w - SplitDec,actualEditor->xoff,y,FGPEN);
    }
    else
    {
      /* if no  line => erase in window: */
      struct RastPort *rp = actualEditor->rp;

      SetAPen(rp,BGPEN);
      RectFill(rp,(ULONG)actualEditor->xoff,
               (ULONG)y,
               (ULONG)actualEditor->xscr,
               (ULONG)y + actualEditor->ch - 1);
      SetAPen(rp,FGPEN);
    }
}
/*#ENDFD*/
/*#FOLD: printAll */
/*********************************
 *
 * printAll
 *
 * print new screen.
 *
 *********************************/

void printAll()
{
  register UWORD y,ch,count;
  register struct Zline **z;
  register struct RastPort *rp;

  y     = actualEditor->yoff;
  ch    = actualEditor->ch;
  count = actualEditor->wch;
  rp    = actualEditor->rp;

  /* Output the line: */
  for (z = actualEditor->zlinesptr; count-- && (*z != NULL);
      z++, y += ch)
    printLine(*z,y);

  /* erase right and bottom border: */
  SetAPen(rp,BGPEN);
  if (actualEditor->yru >= y)
    RectFill(rp,(ULONG)actualEditor->xoff,(ULONG)y,
             (ULONG)actualEditor->xrr,(ULONG)actualEditor->yru);
  if ((actualEditor->xrr >= actualEditor->xrl)
    && (y > actualEditor->yoff))
    RectFill(rp,(ULONG)actualEditor->xrl,(ULONG)actualEditor->yoff,
             (ULONG)actualEditor->xrr,(ULONG)y-1);
  SetAPen(rp,FGPEN);
}
/*#ENDFD*/
/*#FOLD: cn_utoa */
/*******************************
 *
 * cn_utoa(buf,val,len):
 *
```

```
 * Convert UWORD in ASCII.
 *
 * buf ^ buffer for ASCII.
 * val = Value
 * len = Maximum Lenght.
 *
 ****************************/

void cn_utoa();

#asm

  cseg
  public _cn_utoa

_cn_utoa:

  ; 4(sp) ^ Buffer,
  ; 8(sp) = Value,
  ;10(sp) = maximum Lenght.

  lea     4(sp),a0
  move.l  (a0)+,a1
  moveq     #0,d0
  move.w  (a0)+,d0        ; = Value
  move.w  (a0),d1      ; = maximum Lenght
  lea     0(a1,d1.w),a0 ; ^ End of Buffers
  bra.s     .in

.lp:
  divu    #10,d0
  swap    d0
  add.b   #'0',d0      ; d0.w = VAlue MOD 10
  move.b  d0,-(a0)
  clr.w   d0
  swap    d0       ; Zero-Flag is 1, when d0.l = 0
.in:
  dbeq    d1,.lp       ; solange bis Wert = 0 oder String voll
  beq.s   .ib      ; Value = 0 => Rest is full of blanks
  bra.s   .r       ; String full!

.bl:
  move.b  #' ',-(a0)
.ib:
  dbra    d1,.bl
.r:
  rts

#endasm
/*#ENDFD*/
/*#FOLD: printXpos */
/****************************
 *
 * printXpos:
 *
 * Give Xpos of cursor.
 *
 ****************************/

void printXpos()
{
  register struct TextFont *oldFont;
```

```
  register struct Editor *ae = actualEditor•

  cn_utoa(ae->gi_Text + INFO_XPOS_INC,ae->xpos,(UWORD)INFO_XPOS_LEN);

  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_xpos,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text + INFO_XPOS_INC,(ULONG)INFO_XPOS_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
/*#FOLD: printYpos */
/*****************************
 *
 * printYpos:
 *
 * Give Ypos of cursor.
 *
 *****************************/

void printYpos()
{
  register struct TextFont *oldFont;
  register struct Editor *ae = actualEditor;

  cn_utoa(ae->gi_Text + INFO_YPOS_INC,ae->ypos,(UWORD)INFO_YPOS_LEN);

  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_ypos,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text + INFO_YPOS_INC,(ULONG)INFO_YPOS_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
/*#FOLD: printNumLines */
/*****************************
 *
 * printNumLines:
 *
 * Give number of lines
 * of text
 *
 *****************************/

void printNumLines()
{
  register struct TextFont *oldFont;
  register struct Editor *ae = actualEditor;

  cn_utoa(ae->gi_Text + INFO_NUMOFLINES_INC,ae->num_lines,
        (UWORD)INFO_NUMOFLINES_LEN);

  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_numzlines,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text + INFO_NUMOFLINES_INC,
      (ULONG)INFO_NUMOFLINES_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
/*#FOLD: printFold */
/*******************************
```

```
 *
 * printFold:
 *
 * Gives  minfold and maxfold display.
 *
 *******************************/

void printFold()
{
  register struct TextFont *oldFont;
  register struct Editor *ae = actualEditor;

  cn_utoa(ae->gi_Text + INFO_MINFOLD_INC,ae->minfold,
        (UWORD)INFO_MINFOLD_LEN);
  cn_utoa(ae->gi_Text + INFO_MAXFOLD_INC,ae->maxfold,
        (UWORD)INFO_MAXFOLD_LEN);

  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_minfold,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text + INFO_MINFOLD_INC,(ULONG)INFO_MINFOLD_LEN);
  Move(ae->rp,ae->ip_maxfold,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text + INFO_MAXFOLD_INC,(ULONG)INFO_MAXFOLD_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
/*#FOLD: printFlags */
/*****************************
 *
 * printFlags:
 *
 * At Ypos of cursor.
 *
 *****************************/

void printFlags()
{
  register struct TextFont *oldFont;
  register struct Editor *ae = actualEditor;
  register UBYTE *fs = ae->gi_Text + INFO_FLAGS_INC;

  if (ae->insert)
    *fs++ = 'I';
  else
    *fs++ = 'O';
  if (ae->changed)
    *fs++ = 'C';
  else
    *fs++ = 'c';
  if (ae->autoindent)
    *fs++ = 'A';
  else
    *fs++ = 'a';
  if (ae->tabs)
    *fs++ = 'T';
  else
    *fs++ = 't';
  if (ae->skipblanks)
    *fs = 'B';
  else
    *fs = 'b';
```

```
  oldFont = ae->rp->Font;
  SetFont(ae->rp,infoFont);
  Move(ae->rp,ae->ip_flags,ae->ip_topedge);
  Text(ae->rp,ae->gi_Text + INFO_FLAGS_INC,(ULONG)INFO_FLAGS_LEN);
  SetFont(ae->rp,oldFont);
}
/*#ENDFD*/
/*#FOLD: printInfo */
/*****************************
 *
 * printInfo:
 *
 * Gives new  Infoline.
 *
 *****************************/

void printInfo()
{
  printXpos();
  printYpos();
  printNumLines();
  printFold();
  printFlags();
}
/*#ENDFD*/
```

Version 6 Memory.c

```
/*************************************
 *                                   *
 * Module: Memory
 *                                   *
 * Organize memory handling.
 *                                   *
 *************************************/

/*#FOLD: Includes */
/************
 *
 * Includes:
 *
 ************/

#include <exec/types.h>
#include <intuition/intuition.h>
#include <exec/memory.h>
#include "src/Editor.h"
/*#ENDFD*/
/*#FOLD: Defines */
/***********
 *
 * Defines:
 *
 ***********/

#define BIGBLOCK (10240L-sizeof(struct Memoryblock))
#define BLOCKSIZE (5120L-sizeof(struct Memoryblock))
/*#ENDFD*/
/*#FOLD: External Functions */
/**********************
 *
 * External Functions:
 *
 **********************/

struct Memoryblock *AllocMem();
void NewList(),AddTail(),Remove(),FreeMem();
/*#ENDFD*/
/*#FOLD: External Variables */
/*********************
 *
 * External Variables:
 *
 *********************/

extern struct Editor *actualEditor;
/*#ENDFD*/

/***************
 *     *
 * Functions: *
 *     *
 ***************/
```

```
/*#FOLD: freeMemoryblock */
/*****************************************
 *
 * freeMemoryblock(blk)
 *
 * frees the memory of
 * Memoryblocks, without regard for
 * lines that lie in it.
 *
 * blk - Memoryblock.
 *
 *****************************************/

void freeMemoryblock(blk)
struct Memoryblock  *blk;
{
  Remove(blk);
  FreeMem(blk,blk->length + sizeof(struct Memoryblock));
}
/*#ENDFD*/
/*#FOLD: newerBlock */
/*****************************************
 *
 * newerBlock(len):
 *
 * used memory for new memory block with len length
 * and returns the pointer
 * in case of error => NULL.
 *
 * len = Length of memory block.
 *
 *****************************************/

struct Memoryblock *newerBlock(len)
ULONG                    len;
{
  register struct Memoryblock *blk;
  register struct Memorysection *spst;

  /* Allocate memory: */
  if (blk = AllocMem(len+sizeof(struct Memoryblock),MEMF_CLEAR))
  {
    /*Initialize memory block structure: */
    blk->length = len;
    blk->free = len - sizeof(struct Memorysection);
    NewList(&(blk->freeliste));
    spst = (struct Memorysection *)(blk + 1);
    AddTail(&(blk->freeliste),spst);
    spst->len = (UWORD)blk->free;
  }

  return (blk);
}
/*#ENDFD*/
/*#FOLD: searchMemorysection */
/*****************************************
 *
 * searchMemorysection(block,len)
 *
 * Searches for memory piece with len length in the block block
 * and returns the pointer or zero in case
 * of no room.
```

```
 *
 * block ^ Memoryblock.
 * len = Length of line (even).
 *
 ********************************************/

struct Memorysection *searchMemorysection(block,len)
struct Memoryblock                *block;
UWORD                              len;
{
  register UWORD minl,l;
  register struct Memorysection *st,*fnd = NULL;

  /* Now searches through if enough memory: */
  if ((ULONG)len <= block->free)
  {
    /* Minimal length in case found Length <> len */
    minl = len + sizeof(struct Memorysection) + 2;

    /* Searches through list: */
    for (st = block->freeliste.head; st->succ; st = st->succ)
      if ((l = st->len) == len)
      {
        /* Optimal size found: */
        fnd = st;
        break;
      }
      else
        if ((l >= minl) && (l > ((fnd)? fnd->len : 0)))
          /* Search for largest block: */
          fnd = st;
  }

  return (fnd);
}
/*#ENDFD*/
/*#FOLD: ConvertSpstToZline */
/*******************************************
 *
 * ConvertSpstToZline(blk,st,len)
 *
 * Converts memory piece in line.
 * If the piece was longer than the line,
 * the piece is shortened, otherwise the piece is removed
 * from the list.
 *
 * blk ^ memory block of memory piece.
 * st ^ memory piece.
 * len = Length of the line
 *
 ********************************************/

struct Zline *ConvertSpstToZline(blk,st,len)
struct Memoryblock          *blk;
register struct Memorysection   *st;
register UWORD                   len;
{
  register UWORD l;
  register struct Zline *z;

  z = (struct Zline *)st;
```

```
  if (st->len == (l = EVENLEN(len)))
    /* Found piece fits exactly: */
    Remove(st);
  else
  {
    /* distribute piece: */
    ((UBYTE *)st) += (l += sizeof (struct Zline));
    ( st->succ = ((struct Memorysection *)z)->succ )->pred = st;
    ( st->pred = ((struct Memorysection *)z)->pred )->succ = st;
    st->len = ((struct Memorysection *)z)->len - l;
  }
  blk->free -= l;

  z->flags  = 0;
  z->len    = len;

  return (z);
}
/*#ENDFD*/
/*#FOLD: getZline */
/*********************************************
 *
 * getZline(len)
 *
 * Tries to get line with len length and,
 * searches through all of the memory blocks.
 * returns pointer to line or
 * zero if no room.
 *
 * len = Length of the line.
 *
 *********************************************/

struct Zline *getZline(len)
UWORD              len;
{
  register UWORD l;
  register struct Memoryblock *blk,*fblk = NULL;
  register struct Memorysection *st,*fst = NULL;

  /* make even length: */
  l = EVENLEN(len);

  /* run through list of blocks: */
  for (blk = actualEditor->block.head; blk->succ; blk = blk->succ)
    if (st = searchMemorysection(blk,l))
      if (st->len == l)
      {
        /* found optimal memory piece: */
        fblk  = blk;
        fst = st;
        break;
      }
      else
        if (st->len > ((fst)? fst->len : 0))
        {
          /* searches for largest memory piece: */
          fblk  = blk;
          fst = st;
        }

  if (fst)
```

```
    return (ConvertSpstToZline(fblk,fst,len));
  else
    /* no piece found: */
    return (NULL);
}
/*#ENDFD*/
/*#FOLD: optimalBlock */
/*******************************************
 *
 * optimalBlock(block)
 *
 * Searches through all of the memory pieces of the block
 * and makes one out of all of the memory pieces
 * that lie one after another.
 *
 * block ^ Memoryblock.
 *
 *******************************************/

void optimalBlock        (blk)
register struct Memoryblock *blk;
{
  register struct Memorysection *st1,*st2,*st1e;

  for (st1 = blk->freeliste.head; st1->succ; st1 = st1->succ)
  {
    /* Certain pointer directly behind st1: */
    st1e = (struct Memorysection *)
         (((UBYTE *)st1) + st1->len + sizeof(struct Memorysection));

    for (st2 = blk->freeliste.head; st2->succ; st2 = st2->succ)
      if (st1e == st2)
      {
        /* memory piece st2 lies behind st1: */
        st1->len += st2->len + sizeof(struct Memorysection);
        blk->free += sizeof(struct Memorysection);
        Remove(st2);
        st1e = (struct Memorysection *)
             (((UBYTE *)st1)+st1->len+sizeof(struct Memorysection));

        /* begin at the beginning of the list: */
        st2 = (struct Memorysection *) &(blk->freeliste.head);
      }
  }
}
/*#ENDFD*/
/*#FOLD: garbageCollection */
/*****************************************
 *
 * garbageCollection(len)
 *
 * cleans the memory until at least
 * len bytes are free. returns pointer.
 * to memory block where there is room
 * or zero if no room.
 * zero if no room.
 *
 * len = number of bytes that should be accomodated
 *
 *
 *****************************************/
```

```
struct Memoryblock *garbageCollection(len)
UWORD                              len;
{
  struct Memoryblock *blk;
  struct Memorysection *st,spst;
  register UBYTE *ptr,*end;
  register struct Zline *z,*fz;
  register UWORD n;
  register struct Zline **zptr;

  for (blk = actualEditor->block.head; blk->succ; blk = blk->succ)
    /* Only garbage collection if more than one memory piece! */
    if ((st = blk->freeliste.head->succ)? st->succ : FALSE)
    {
      /* test for beginning and end of block memory: */
      ptr = (UBYTE *)(blk + 1);
      end = ptr + blk->length;

      /* lines lie at the beginning: */
      while (ptr < end)
      {
        /* searches for lines that lie at next ptr: */
        fz = NULL;
        for (z = actualEditor->zlines.head; z->succ; z = z->succ)
          if ((z >= ptr) && (z < end))
            if ((z < fz) || (fz == NULL))
              fz = z;

        if (fz)
          if (fz != ptr)
          {
            /* note old pointer: */
            z = fz;
            st = (struct Memorysection *)ptr;
            spst = *st;

            /* writes line after ptr: */
            for (n = (EVENLEN(fz->len) + sizeof(struct Zline)) >> 1;
                n; n--, ((UWORD *)ptr)++, ((UWORD       *)fz)++ )
              *((UWORD  *)ptr) =  *((UWORD       *)fz);

            /* pointer correction: */
            fz = (struct Zline *)st;
            fz->succ->pred = fz;
            fz->pred->succ = fz;
            if (actualEditor->actual == z)
              actualEditor->actual = fz;
            if  (actualEditor->lineptr == z)
              actualEditor->lineptr =   fz;
            for (n =  0,  zptr = actualEditor->zlinesptr;
                n <= actualEditor->wch; n++, zptr++)
              if (*zptr == z)
              {
                *zptr = fz;
                break;
              }

            st = (struct Memorysection *)ptr;
            st->succ = spst.succ;
            st->pred = spst.pred;
            st->len = spst.len;
            spst.succ->pred = st;
```

```
            spst.pred->succ = st;

            /* push memory pieces together: */
            optimalBlock(blk);
          }
          else
            /* set ptr high: */
            ptr += EVENLEN(fz->len) + sizeof(struct Zline);
        else
          /* no more lines in block! */
          break;
      }   /* of while */

      /* Test if there is enough memory: */
      if (searchMemorysection(blk,len))
        return (blk);
    }   /* of if (at least 2 blocks) */

  return (NULL);
}
/*#ENDFD*/
/*#FOLD: newZline */
/*****************************************
 *
 * newZline(len)
 *
 * get room for new line,
 * fits characters in len.
 * returns pointer to new line,
 * or zero if error.
 *
 * len = number of characters.
 *
 *****************************************/

struct Zline *newZline(len)
UWORD             len;
{
  register struct Zline *z;
  register struct Memoryblock *blk;
  register struct Memorysection *st;
  register UWORD l;

  if (actualEditor->block.head->succ)
    /* blocks already exist: */
    if (z = getZline(len))
      return (z);
    else
      /*  Garbage-Collection:line not directly usable; */
      if (blk = garbageCollection(l = EVENLEN(len)))
        if (st = searchMemorysection(blk,l))
          return (ConvertSpstToZline(blk,st,len));
        else
          /* may not actually be encountered !!! */
          return (NULL);
      else
        /* no reults for Garbage-Collection ->pick new block */
        if (blk = newerBlock(BLOCKSIZE))
        {
          AddTail(&(actualEditor->block),blk);
          return (getZline(len));
        }
```

```
        else
          return (NULL);
    else
      /* no block in the block list: */
      if (blk = newerBlock(BIGBLOCK))
      {
        AddTail(&(actualEditor->block),blk);
        return (getZline(len));
      }
      else
        return (NULL);
}
/*#ENDFD*/
/*#FOLD: deleteZline */
/*******************************************
 *
 * deleteZline(line)
 *
 * Delete line from the list of lines.
 * The occupied memory word in the memory
 * section is changed and the corresponding
 * memory block optimized.
 *
 * line ^ line.
 *
 *******************************************/

void deleteZline      (line)
register struct Zline *line;
{
  register struct Memoryblock *blk,*fblk = NULL;
  register struct Memorysection *st;
  register struct Zline **zptr;
  register UWORD n;

  /* remove line from list: */
  Remove(line);

  /* determine in which block the line lies: */
  for (blk = actualEditor->block.head; blk->succ; blk = blk->succ)
    if ((line > blk) && (line < (((UBYTE *)(blk + 1)) + blk->length)))
    {
      fblk = blk;
      break;
    }

  if (fblk)
  {
    /* convert line into memory peices, insert in list: */
    fblk->free += ( ((struct Memorysection *)line)->len
                = EVENLEN(line->len) );
    AddTail(&(fblk->freeliste),line);

    optimalBlock(fblk);
    if (fblk->length == (fblk->free + sizeof(struct Memorysection)))
      /* Block contains no more lines: Delete! */
      freeMemoryblock(fblk);
  }
}
/*#ENDFD*/
```

Version 6 Command.c

See Section 4.3.9 for a complete listing of the `comand.c` module.

Index

Optional Diskette

For your convenience, the program listings contained in this book are available on an Amiga formatted floppy diskette. You should order the diskette if you want to use the programs, but don't want to type them in from the listings in the book.

All programs on the diskette have been fully tested. You can change the programs for your particular needs. The diskette is available for $14.95 plus $2.00 ($5.00 foreign) for postage and handling.

When ordering, please give your name and shipping address. Enclose a check, money order or credit card information. Mail your order to:

Abacus
5370 52nd Street SE
Grand Rapids, MI 49512

Or for fast service, call **616-698-0330.**
Credit Card orders only **1-800-451-4319.**

New Software

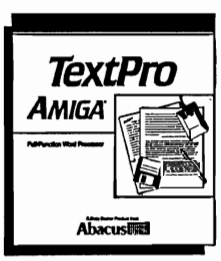
All Abacus software runs on the Amiga 500, Amiga 1000 or Amiga 2000. Each package is fully compatible with our other products in the Amiga line

The *Ideal* AMIGA wordprocessor

TextPro
AMIGA

TextPro AMIGA upholds the true spirit of the AMIGA: it's powerful, it has a surprising number of "extra" features, but it's also very easy to use. **TextPro AMIGA**—the Ideal AMIGA word processor that proves just how easy word processing can be. You can write your first documents immediately, with a minimum of learning—without even reading the manual. But **TextPro AMIGA** is much more than a beginner's package. Ultra-fast onscreen formatting, graphic merge capabilities, automatic hyphenation and many more features make **TextPro AMIGA** ideal for the professional user as well. **TextPro AMIGA** features:

- High-speed text input and editing
- Functions accessible through menus or shortcut keys
- Fast onscreen formatting
- Automatic hyphenation
- Versatile function key assignment
- Save any section of an AMIGA screen & print as text
- Loading and saving through the RS-232 interface
- Multiple tab settings
- Accepts IFF format graphics in texts
- Extremely flexible printer adaptations. Printer drivers for most popular dot-matrix printers included
- Includes thorough manual
- Not copy protected



TextPro AMIGA sets a new standard for word processing packages in its price range. So easy to use and modestly priced that any AMIGA owner can use it—so packed with advanced features, you can't pass it up.

Suggested retail price: **$79.95**

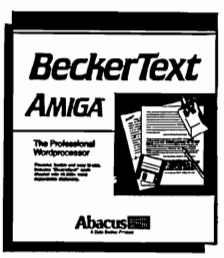
More than word processing...

BeckerText
AMIGA

This is one program for serious AMIGA owners. **BeckerText Amiga** is more than a word processor. It has all the features of **TextPro AMIGA**, but it also has features that you might not expect:

- Fast WYSIWYG formatting
- Calculations within a text—like having a spreadsheet program anytime you want it
- Templates for calculations in columns
- Line spacing options
- Auto-hyphenation and Auto-indexing
- Multiple-column printing, up to 5 columns on a single page
- Online dictionary checks spelling in text as it's written
- Spell checker for interactive proofing of documents
- Up to 999 characters per line (with scrolling)
- Many more features for the professional

BeckerText AMIGA is a vital addition for C programmers—it's an extremely flexible C editor. Whether you're deleting, adding or duplicating a block of C source-code, **BeckerText AMIGA** does it all, automatically. And the online dictionary acts as a C syntax checker and finds syntax errors in a flash.



BeckerText AMIGA. When you need more from your word processor than just word processing.

Suggested retail price: **$150.00**

Imagine the perfect database

DataRetrieve
AMIGA

Imagine, for a moment, what the perfect database for your AMIGA would have. You'd want power and speed, for quick access to your information. An unlimited amount of storage space. And you'd want it easy to use—no baffling commands or file structures—with a graphic interface that does your AMIGA justice.

Enter **DataRetrieve AMIGA**. It's unlike any other database you can buy. Powerful, feature-packed, with the capacity for any business or personal application—mailing lists, inventory, billing, etc. Yet it's so simple to use, it's startling. **DataRetrieve AMIGA**'s drop-down menus help you to define files quickly. Then you conveniently enter information using on-screen templates. **DataRetrieve AMIGA** takes advantage of the Amiga's multi-tasking capability for *optimum* processing speed.

DataRetrieve AMIGA features:

- Open eight files simultaneously
- Password protection
- Edit files in memory
- Maximum of 80 index fields with variable precision (1-999 characters)
- Convenient search/select criteria (range, AND/OR comparisons)
- Text, date, time, numeric and selection fields, IFF file reading capability
- Exchange data with other software packages (for form letters, mailing lists, etc.)
- Control operations with keyboard or mouse
- Adjustable screen masks, up to 5000 x 5000 pixels
- Insert graphic elements into screen masks (e.g., rectangles, circles, lines, patterns, etc.)
- Screen masks support different text styles and sizes
- Multiple text fields with word make-up and formatting capabilities
- Integrated printer masks and list editor.
- Maximum filesize 2 billion characters
- Maximum data record size 64,000 characters
- Maximum data set 2 billion characters
- Unlimited number of data fields
- Maximum field size 32,000 characters

DataRetrieve AMIGA—it'll handle your data with the speed and easy operation that you've come to expect from Abacus products for the AMIGA.

Suggested retail price: **$79.95**

Not just for the experts

AssemPro
AMIGA

AssemPro AMIGA lets every Amiga owner enjoy the benefits of fast machine language programming.

Because machine language programming isn't just for 68000 experts. **AssemPro AMIGA** is easily learned and user-friendly—it uses Amiga menus for simplicity. But **AssemPro AMIGA** boasts a long list of professional features that eliminate the tedium and repetition of M/L programming. **AssemPro AMIGA** is the complete developer's package for writing of 68000 machine language on the Amiga, complete with editor, debugger, disassembler and reassembler. **AssemPro AMIGA** is the perfect introduction to machine langage development and programming. And it's even got what you 68000 experts need.

AssemPro AMIGA features:

- Written completely in machine language, for ultra-fast operation
- Integrated editor, debugger, disassembler, reassembler
- Large operating system library
- Runs under CLI and Workbench
- Produces either PC-relocatable or absolute code
- Macros possible for nearly any parameter (of different types)
- Error search function
- Cross-reference list
- Menu-controlled conditional and repeated assembly
- Full 32-bit arithmetic
- Debugger with 68020 single-step emulation
- Runs on any AMIGA with 512K and Kickstart 1.2.

Suggested retail price: **$99.95**

Abacus AmigaDOS® Toolbox

Essential software tools for all Amiga users.

NEW FOR ALL AMIGAS!

A collection of essential, powerful and easy-to-use tools.

The Abacus AmigaDos® ToolBox includes 11 new fonts, new handy CLI commands, a disk copier, floppy disk speeder, screen capture utility and more.

Suggested Retail Price: $59.95

from a company you can count on

Abacus

5370 52nd Street SE • Grand Rapids, MI 49508 • Phone (616) 698-0330 • Telex 709-101 • Facsimile (616) 698-0325

Companion Diskette

For your convenience, the program listings contained in this book are available on an Amiga formatted floppy diskette. You should order the diskette if you want to use the programs, but don't want to type them in from the listings in the book.

All programs on the diskette have been fully tested. You can change the programs for your particular needs. The diskette is available for $14.95 plus $2.00 ($5.00 foreign) for postage and handling.

When ordering, please give your name and shipping address. Enclose a check, money order or credit card information. Mail your order to:

Abacus
5370 52nd Street SE
Grand Rapids, MI 49512

Or for fast service, call **616-698-0330.**
Credit Card orders only **1-800-451-4319.**

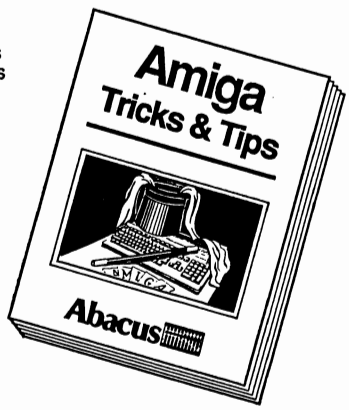
Books for the AMIGA

Amiga Tricks & Tips

Amiga Tricks & Tips follows our tradition of other Tricks and Tips books for CBM users. Presents dozens of tips on accessing libraries from BASIC, custom character sets, AmigaDOS, sound, important 68000 memory locations, and much more!

Topics include:

- Diverse and useful programming techniques
- Displaying 64 colors on screen simultaneously
- Accessing libraries from BASIC
- Creating custom character sets
- Using Amiga DOS and graphics
- Dozens of tips on windows
- Programming aids
- Covers important 68000 memory locations



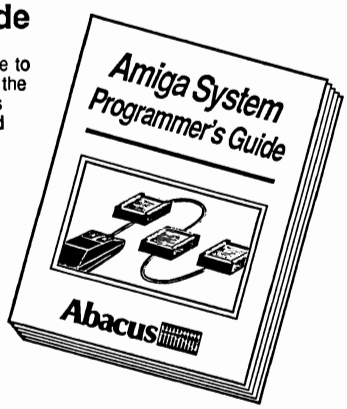
Volume 5 Suggested retail price $19.95 ISBN 0-916439-88-7

Amiga System Programmer's Guide

Amiga System Programmer's Guide is a comprehensive guide to what goes on inside the Amiga in a single volume. Explains in detail the Amiga chips (68000, CIA, Agnus, Denise, Paula) and how to access them. All the Amiga's powerful interfaces and features are explained and documented in a clear precise manner.

Topics include:

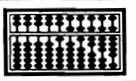
- EXEC Structure
- Multitasking functions
- I/O management through devices and I/O request
- Interrupts and resource management
- RESET and its operation
- DOS libraries
- Disk Management
- Detailed information about the CLI and its commands



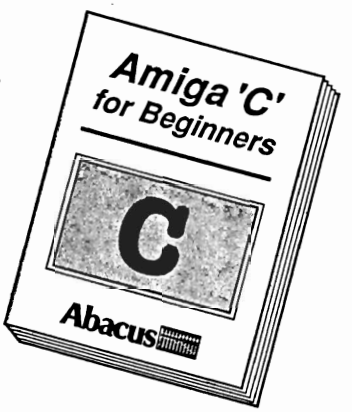
Volume 6 Suggested retail price $34.95 ISBN 1-55755-034-4

Save Time and Money!-Optional program disks are available for all our Amiga reference books (except Amiga for Beginners). All programs listed in the book are on each respective disk and will save you countless hours of typing! $14.95

Books for the AMIGA



Amiga C for Beginners



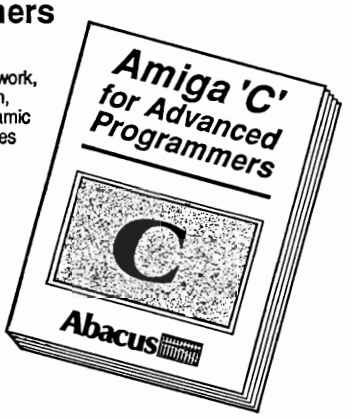
An introduction to learning the popular C language. Explains the language elements using examples specifically geared to the Amiga. Describes C library routines, how the compiler works and more.

Topics include:

- Particulars of C
- How a compiler works
- Writing your first program
- The scope of the language (loops, conditions, functions, structures)
- Special features of C
- Important routines in the C libraries
- Input/Output
- Tricks and Tips for finding errors
- Introduction to direct programming of the operating system (windows, screens, direct text output, DOS functions)
- Using the LATTICE and AZTEC C compilers

Volume 10 Suggested retail price $19.95 ISBN 1-55755-045-X

Amiga C for Advanced Programmers

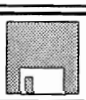


Amiga C for Advanced Programmers- contains a wealth of information from the pros: how compilers, assemblers and linkers work, designing and programming user friendly interfaces using Intuition, managing large programming projects, using jump tables and dynamic arrays, coming assembly language and C codes, and more. Includes complete source code for text editor.

Topics include:

- Using INCLUDE, DEFINE and CASTS
- Debugging and optimizing assembler sources
- All about Intuition programming (windows, screens, pulldown menus, requesters, gadgets)
- Programming the console devices
- A professional editor's view of problems with developing larger programs
- Using MAKE correctly
- Debugging C programs with different utilities
- Folding (formatting text lines and functions for readability)

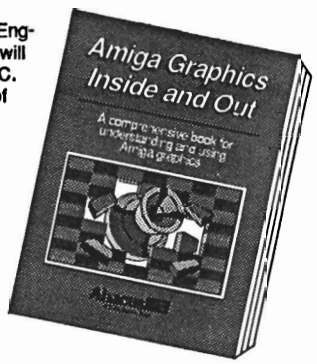
Volume 11 Suggested retail price $24.95 ISBN 1-55755-046-8



Save Time and Money!-Optional program disks are available for all our Amiga reference books (except Amiga for Beginners). All programs listed in the book are on each respective disk and will save you countless hours of typing! $14.95

Books for the AMIGA

Amiga Graphics Inside & Out



The Amiga Graphics Inside & Out book will show you simply and in plain English the super graphic features and functions of the Amiga in detail. You will learn the graphic features that can be accessed from AmigaBASIC or C. The advanced user will learn graphic programming in C with examples of points, lines, rectangles, polygons, colors and more. Amiga Graphics Inside & Out contains a complete description of the Amiga graphic system - View, ViewPort, RastPort, bitmap mapping, screens, and windows.

Topics include:

- Accessing fonts and type styles in AmigaBASIC
- CAD on a 1024 x 1024 super bitmap, Using graphic library routines
- New ways to access libraries and chips from BASIC - 4096 colors at once, color patterns, screen and window dumps to printer
- Graphic programming in C - points, lines, rectangles, polygons,colors
- Amiga animation explained including sprites, bobs and AnimObs, Copper and blitter programming

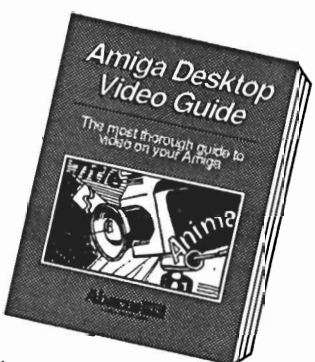
Volume 13 Suggested retail price $34.95 ISBN 1-55755-052-2

Optional Diskette	$14.95	#727

Amiga Desktop Video Guide

The **Amiga Desktop Video Guide** is the most complete and useful guide to desktop video on the Amiga.

Amiga Desktop Video Guide covers all the basics – defining video terms, selecting genlocks, digitizers, scanners, VCRs, camera and connecting them to the Amiga.



Just a few of the topics you'll find described in this excellent book:

- The Basics of Video
- Genlocks
- Digitizers and Scanners
- Frame Grabbers/Frame Buffers
- How to connect VCRs, VTRs, and Cameras to the Amiga
- Animation
- Video Titling
- Music and Videos
- Home Video
- Advanced Techniques
- Using the Amiga to add or incorporate Special Effects to a video
- Tips on Paint, Ray Tracing, and 3-D Rendering in Commercial Applications

Volume 14 • Suggested Retail Price $19.95 • ISBN 1-55755-057-3

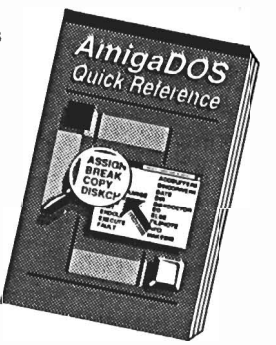
Save Time and Money!-Optional program disks are available for all our Amiga reference books (except Amiga for Beginners and AmigasDOS Quick Reference). Programs listed in the book are on each respective disk and saves countless hours of typing! $14.95

Books for the AMIGA

AmigaDOS Quick Reference Guide

AmigaDos Quick Reference Guide is an easy-to-use reference tool for beginners and advanced programmers alike. You can quickly find commands for your Amiga by using the three handy indexes designed with the user in mind. All commands are in alphabetical order for easy reference. The most useful information you need fast can be found- including:

- All AmigaDOS commands described, including Workbench 1.3
- Command syntax and arguments described with examples
- CLI shortcuts
- CTRL sequences
- ESCape sequences
- Amiga ASCII table
- Guru Meditation Codes
- Error messages with their corresponding numbers



Three indexes for quick information at your fingertips! The AmigaDOS Quick Reference Guide is an indispensable tool you'll want to keep close to your Amiga.

Suggested retail price US $9.95 ISBN 155755-049-2

Abacus Amiga Books

Vol. 1 Amiga for Beginners	1-55755-021-2	$16.95
Vol. 2 AmigaBASIC Inside & Out	0-916439-87-9	$24.95
Vol. 3 Amiga 3D Graphic Programming In BASIC	1-55755-044-1	$19.95
Vol. 4 Amiga Machine Language	1-55755-025-5	$19.95
Vol. 5 Amiga Tricks & Tips	0-916439-88-7	$19.95
Vol. 6 Amiga System Programmers Guide	1-55755-034-4	$34.95
Vol. 7 Advanced System Programmers Guide	1-55755-047-6	$34.95
Vol. 8 AmigaDOS Inside & Out	1-55755-041-7	$19.95
Vol. 9 Amiga Disk Drives Inside & Out	1-55755-042-5	$29.95
Vol. 10 Amiga C for Beginners	1-55755-045-X	$19.95
Vol. 11 Amiga C for Advanced Programmers	1-55755-046-8	$34.95
Vol. 12 More Tricks & Tips for the Amiga	1-55755-051-4	$19.95
Vol. 13 Amiga Graphics Inside & Out	1-55755-052-2	$34.95
Vol. 14 Amiga Desktop Video Guide	1-55755-057-3	$19.95
AmigaDOS Quick Reference Guide	1-55755-049-2	$ 9.95

Amiga® C for Advanced Programmers

Amiga C for Advanced Programmers is for all Amiga programmers who need to understand and increase their productivity sooner using the popular C programming language.

Amiga C for Advanced Programmers contains a wealth of information from the C programming pros: how compilers, assemblers and linkers work, designing and programming user friendly interfaces utilizing the Amiga's built-in user interface Intuition, managing large C programming projects, using jump tables and dynamic arrays, combining assembly language and C codes, and much more.

Amiga C for Advanced Programmers includes the complete source code for a C based text editor. This book gives you easy, straight forward explanations on how C compilers work and will show you how to design your own large C language projects more professionally.

Amiga C for Advanced Programmers shows you examples on how to use the many commands including DEFINE, INCLUDE, CAST and MAKE correctly. If you are an advanced programmer who is acquainted with the features and functions of C, you'll see how to create user-friendly programs accessing the Amiga's built-in user interface, Intuition.

ISBN 1-55755-046-8

9 781557 550460 90000

An advanced guide to programming the Amiga using the C language

You'll see how to debug, optimize and combine assembler source with C code.

Topics included in ***Amiga C for Advanced Programmers:***

- How the Aztec© compiler works (assembly, compiling, linking)
- Using INCLUDE, DEFINE and CAST
- Debugging and optimizing assembler sources
- Jump tables and dynamic arrays in C
- Combining assembler sources with C code
- All about programming Intuition including windows, screens, pulldown menus, requesters, gadgets and more
- Programming the console device
- A professional editor's view of problems with developing larger programs
- Using MAKE correctly - Debugging C programs with different utilities
- Complete source code and instructions for creating your own text editor in C!

Companion Program Diskette available:
Contains every program listed in the book– complete, error-free and ready to run! Saves you hours of typing in the program listings.

Abacus

5370 52nd Street SE, Grand