Introducing ATARI ST machine code

R Pearson
S Hodgson

zzSoft

Introducing ATARI ST machine code

Roger Pearson
Sean Hodgson

First Edition: December 1990
Revised: September 1991

Published by:

zzSoft
25 Honeyhole
Blackburn
Lancashire
England
BB2 3BQ

ISBN 1 873423 01 2

Table of Contents

Introduction

This book aims to teach the practical fundamentals of using MC 68000 assembly language on the ATARI ST range of personal computers. No knowledge of assembly language is assumed, through is is expected that the reader has the use of an ATARI ST and has the ability to copy, move and delete files, format disks, etc from the GEM desktop, and use the GEM interface. It is not practical to study a programming language without first acquiring these basic skills. Much theory has been omitted – as this is a practical book, and many aspects of assembly language programming have had to be left out lest the book became too long and unmanageable.

This book was initially called 'A Practical Introduction to ATARI ST Assembly Language' and in most respects it still fits that title. Game programmers will not find the secrets to the next mega zapping game, as the book is squarely aimed at the ST application programmer. This is not to say that the games programmer will not learn anything but the book is not specific to his/her subject. By application programming is meant those utilities and programs that are of a more serious nature eg GEM applications. However, as the book is an introduction there is no lengthy source code listing of a large application. Rather the book details some of the methods and means of achieving such a task but does not tackle any lengthy programming project it self. By studying the book and using the disk the reader should at the end of the book, if he or she has studied diligently and most importantly patiently, be able to program in assembly language with some degree of proficiency. Naturally not every aspect of the programmers art could be covered and very much more study and practise will be needed by the student to be able to write full scale and complex code. It is sincerely hoped by the author that this book will be useful in taking the reader on this path. Others who read this book as part of a hobby will find much of interest and it is hoped that they too will gain benefit from this book.

As many readers may have the excellent and increasingly popular GFA BASIC a chapter has been devoted to using assembly language from it.

Programming style

The author's own style of programming may be quite different from yours or indeed anyone else's. It should be noted that reading (and trying to understand) another programmers source code is not always easy or desirable as someone else's code may complicate what you perceive as a simple problem. On the other hand what programmer hasn't learnt much from the work and source code of others? Programming is as much creative work as it is an exact and demanding science and as such a word processor written by one person would undoubtedly be very much different in programming style and the resultant operation of the program. Also it is only fair to warn the novice that the road to proficient and bug free (is it possible!) programming is fraught with many days and (mostly) nights of frustration and the general pulling–out of hair (one's own mostly!).

The disk

This book comes with a double–sided 3.5″ floppy disk which has a symbolic assembler/linker, symbolic disassembler or debugger, a resource construction program and other programming utilities on it. Each programming example given in this book is also on the disk as an executable file (a file with the extension .PRG) and as source code (a file with the extension .S) and resource file (.RSC), if any. If you have a single–sided disk drive the disk should be returned directly to zzSoft for an immediate replacement. Please include two first class stamps, and your name and address.

The disk and all programming examples can be run on any ATARI ST: STE, STFM; 520, 1040, and megas. Every programming example can be assembled and run without having to first go out and buy an assembler. The assembler and debugger can also be used to develop and debug your own programs too. However as many people use (as I do), HiSofts' DEVPAC (DEVelopment PACkage) every example program can be easily adapted to be used by this software.

Even at this stage the beginner may well feel lost with mention of such technical words as 'debugger', 'source code'. Don't worry each chapter will carefully explain the practical use of these words in real–life programming examples. But also see the glossary for a short explanation of these programming terms.

If you are not familiar with using a text editor please read the chapter 'Using the Text Editor' as all programming should be carried out from the text editor.

Backup the disk!

You should make a backup copy of the supplied disk before you do anything else. If you inadvertently manage to damage the disk, or if the disk was in some way damaged when you received it, or you deleted some files by mistake you should return the original disk to zzSoft for replacement. Please enclose two first class stamps and your name and address.

What's on the disk ?

zzSoft's text editor: The text editor, EDITOR.PRG, is specially designed to enable the reader to get into assembler programming with the minimum of fuss. You can 'assemble' and 'debug' all the example programs given in the book directly from the text editor. For speed of execution and ease of use the assembler and debugger should be run from a RAM disk.

Note that it is not absolutely necessary that the assembler and debugger be run from a RAM disk, but it does speed up the process of assembling. If you want to run the assembler and debugger from a floppy disk then you should remove the RAM disk program from the AUTO folder.

The main items on the disk are:

Editor	EDITOR.PRG
Assembler and linker:	ASSEMBLR.TTP, LINK.TTP
Debugger	DEBUG.TTP
Resource construction kit	MKRSC.PRG

GEMDOS, BIOS, XBIOS, AES and VDI libraries

Example source code, executable files, etc

To use the disk

Remove all files except assembler, linker, debugger, and source code from back-up disk. You should then double-click on EDITOR.PRG. Please see READ_ME.1ST file on the disk for more details on how to use the disk in the most appropriate manner for your computer set-up.

READ_ME.1ST

This ASCII text file is on the supplied disk and can be loaded into the supplied text editor and viewed there. Alternatively, it may be better to print it out as it contains the very latest information about the book and disk that could not appear in the book. It is possible to print it out from the text editor. It is important that you read this file carefully.

Copyright and help notice

Please remember that although the book gives many examples of assembly language source code which are also duplicated on the supplied disk once you alter the source code and possibly run into difficulty zzSoft cannot help. Help is limited to the source code as given on the disk. When you write your own assembly language programs or alter any of the source code on the disk YOU ARE ON YOUR OWN! zzSoft, its agents, its distributors or its retailers cannot help, nor give advice as to how to write your own programs.

Help and advice from zzSoft is strictly limited to the book and source code and its operation within zzSoft's text editor.

However, any resultant object code is free from any such restriction.

The assembler is for use with the book and source code on the companion disk and is not guaranteed for any other purpose. It is obviously possible to develop one's own applications using it but zzSoft do not in any way guarantee it fit for such a purpose.

Copyright: debugger and MKRSC.PRG (Resource Construction program)

The debugger on the supplied disk is in the public domain, as is the resource construction program (RCP) or kit (MKRSC.PRG) – with some limitations as to its distribution. However, permission has been granted by the copyright owner of the resource construction kit to include it on the disk and to include instructions for its use in this book. The debugger from Sozobon has no restrictions as to its use, except that Sozobon be acknowledged. Source code for the debugger, written in C, is in the public domain and freely available from many PD libraries and BBS's.

Acnowledgements

Thanks to Alistair Bodin, Prosupport manager, Atari UK for his prompt attention to all enquiries.

J Charlton, of Winnipeg, Canada for allowing the authors the use of his excellent resource contruction kit.

However, any resultant object code is free from any such restriction.

The assembler is for use with the book and source code on the companion disk and is not guaranteed for any other purpose. It is obviously possible to develop one's own applications using it but zzSoft do not in any way guarantee it fit for such a purpose.

Copyright: debugger and MKRSC.PRG (Resource Construction program)

The debugger on the supplied disk is in the public domain, as is the resource construction program (RCP) or kit (MKRSC.PRG) – with some limitations as to its distribution. However, permission has been granted by the copyright owner of the resource construction kit to include it on the disk and to include instructions for its use in this book. The debugger from Sozobon has no restrictions as to its use, except that Sozobon be acknowledged. Source code for the debugger, written in C, is in the public domain and freely available from many PD libraries and BBS's.

Acknowledgements

Thanks to Alistair Bodin, Press support manager, Atari UK for his prompt attention to all enquiries.

J Charlton, of Winnipeg, Canada for allowing the authors the use of his excellent resource construction kit.

Chapter 1
Starting Out

This chapter introduces the reader to a simple assembly language program, and the terms of reference used in its operation.

Why assembly language when there are so many excellent higher level languages available for the ST, like GFA BASIC? The most usual answer to this question is speed. Assembly language programs, efficiently written, undoubtedly run faster than programs written in higher level languages such as 'C' and BASIC. But why is speed essential? For games, screen handling of text, interfacing to peripherals, etc speed is very important. Poor scrolling speed, slow screen updates soon become very annoying. The faster the program the more efficient, and effective it is seen by the user. And this sells software.

Assembly language is almost as low as a programming language can get. In the hierarchy of languages the more English-like the language the higher up it is on the ladder. BASIC for example allows the printing to screen of a line of text (or 'string' of text) with the command 'print "string"', whilst using assembly language involves a great deal more than that as we will see. Printing text to the screen cannot be invoked by a simple 'print' command in assembler.

Text editor

To write or develop a computer program a text editor is needed to enter the source code. Source code consists of the text which we write which is finally assembled. The source code then becomes object code, which is usually an executable or runnable file, ie one that can be double-clicked or run from the desktop. Sometimes text editors are an integral part of an assembler/debugger, as DEVPAC 2, and zzSoft's is. On the supplied disk is zzSoft's text editor specially written for this book. When an assembly language program is written it is then 'passed' to the assembler, which is another program that converts the text we have written into an executable file, ie one that we can double-click. A disassembler or debugger is another specially written program that allows us to examine the executable file step-by-step if necessary. A debugger is an essential part of a programmers equipment as assembly language programs often don't work first time! Assembly language is very error prone and often the only way to find a fault, or bug, in the program, is to load it into a debugger and by the process of examining the part where the fault takes place try to locate the bug. It can then be corrected in the source code and then reassembled. Some bugs can be very hard to find, and many hours can be spent searching for a particularly elusive bug.

The process of developing and testing assembly language program's can be shown like this:

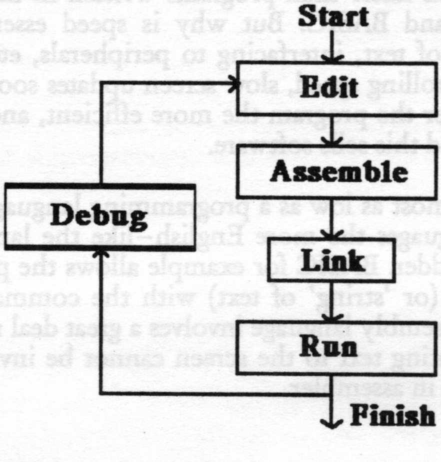

diagram 1:1

GEM and the operating system

A lot of programs written for the ST utilise the GEM (Graphics Environment Manager) interface, which consists of Windows, Icons, Menus, Pointer (mouse) or the WIMP interface, although most games do not. GEM (which is a computer program held in the ST's ROM) itself rests upon a hierarchy of other programs. These programs (also resident in the ST's ROM) which make up the ST's operating system (o/s), ensure that disk, screen, keyboard, and other peripherals can be easily accessed by the programmer. It would be an enormous task if every program had to have its own file loading program, screen interface, etc. Although many programmers do write special routines that bypass or improve on the internal ROM functions eg Tempus 2, a particularly fast text editor.

GEM consists primarily of the VDI (Virtual Display Interface)– which is mainly concerned with text, and graphics, etc, the AES (Application Environment Service) which handles windows, drop down menus, etc. Lower down are the internal functions that handle files, the keyboard, etc, and consists of the BIOS (Basic Input/Output System), the XBIOS (EXtended Basic Output/Input System), GEMDOS (GEM Disk Operating System) and finally the A–line, which has some very fast routines that are not particularly well documented. The whole range of services come under the general heading of the operating system (o/s), which for the ST is called TOS. **T**he **O**perating **S**ystem.

It is possible to access all of the parts of the GEM environment using assembly language, and many examples will be given as well as an example of opening, moving, and resizing a GEM window. Menus, dialog boxes, and mouse handling will also be looked, etc.

Our first program:

This program is about as small and as elementary as possible, but it illustrates many fundamentals, and although it does not do much it is as good as any place to start. To begin either enter the following source code in the text editor or preferably load it straight into the editor from the supplied disk. *This procedure applies to all the example programs in the book.* Programs are executed in an orderly fashion starting at line one and proceeding to the next line unless instructed by the program to go somewhere else in the program. Note that unlike many BASIC's no line numbers are used and only one 68000 statement per line is allowed.

```
* EX1.S This short program (or source code when referring to the
* actual text) prints the letter 'A' on the screen. Then it waits for a
* key press and and then exits back to the desktop (if it is run from
* the desktop) or the text editor in an orderly fashion.

start:  move.w  #65,-(sp) ; start of program. #65="A"
        move.w  #2,-(sp)
        trap    #1
```

```
        addq.l    #4,sp   ; correct stack

wait:   move.w    #1,-(sp)  ; wait for a key press so we can see letter
* on screen
        trap      #1
        addq.l    #2,sp     ; correct stack

exit:   move.w    #20,-(sp) ; leave gracefully!
        move.w    #$4c,-(sp)
        trap      #1
* exit from program properly
```

Well, what does all this mean? First things first and we need to look at how each line of text is processed by the assembler. The format is shown below:

Label Mnemonic Operands(s) Comment

For example,

```
start:    move.w    #65-,(sp)    ; start of program
```

Label

A label is a marker that helps the programmer to navigate his way around a program or source code and allows the naming of subroutines to have a useful meaning. Also, you will see later that labels are used to refer to an address by reference to the label. Thus in the above example 'start:' helps us to recognize that we are at the start of the program; not particularly useful but if we ever wanted to go back there it is possible to do so by reference to the 'start:' label. If the program was to become a subroutine then we may rename the label to 'wait_for_a_char:' or whatever and then reference it as such.

Using labels with meaningful names is a prerequisite of good programming practise. Also, importantly, labels can be included in the assembled program and the executable program then can be loaded into a debugger and examined using the labels as 'signposts'. See chapter three for a practical example. Labels help immensely in this process. Because the assembler can accept labels and symbols (labels and symbols are used synonymously) the assembler is called a symbolic assembler. A debugger that can accept and use labels is called a symbolic debugger. A debugger is sometimes called a monitor or disassembler. The assembler and debugger with the disk provided with this book are both symbolic.

Note that labels must be followed by a colon ':' and that all text including labels should be lower-case. Other assemblers do not expect a colon after a label, or symbol. A label may start with any character or underline '__' and be followed by underlines, digits, 0-9, or a full stop '.' or period.

These are acceptable labels

my_label:
___another_label:
_1_:

but these are not

9nth_label:
:_label:

Mnemonic

This consists of 68000 assembler instructions, and in the example, the first line mnemonic is 'move.w'. Some 68000 instructions need a size specifier, which can be '.b', '.w'. or '.l'. These are short for byte, word, and long (often called a long word). If a size extension is expected and none is given eg 'move' then the assembler assumes 'move.w' is the required extension. This is the usual practise adopted by most if not all assemblers.

Operands

This field holds the registers, or symbols that are acted upon, eg '#1,-(sp)'. This means, taken in conjunction with the mnemonic, decrement the address held by the stack pointer (sp or register a7) then move or place 65 (decimal) in the place referred to by the stack register, ie the address pointed to by the sp. This will be described more thoroughly later.

Comment

All comments, or remarks must be preceded by a ';' (semicolon) or if starting on a new line in the label field a '*' or ';'. It is always advisable to comment your source code. You may understand what it means now, but what about in three months time! For your own mental health '*comment your source code!*'.

Assembling the program

If you press ALT–A or go to the drop down menu named 'Program' and select *Assemble...* then the following dialog box – see diagram 1:2 – will appear. Pressing the Return key or clicking in the 'Assemble' button will invoke the assembler, and linker. After various messages have appeared advising of correct assembly and linking, you can then go back to the text editor and now you can 'run' the program and see what is does for yourself. Press ALT–X to run the program or select *Run* from the *Program* drop down menu. Please see chapter twenty four for more details on using the assembler.

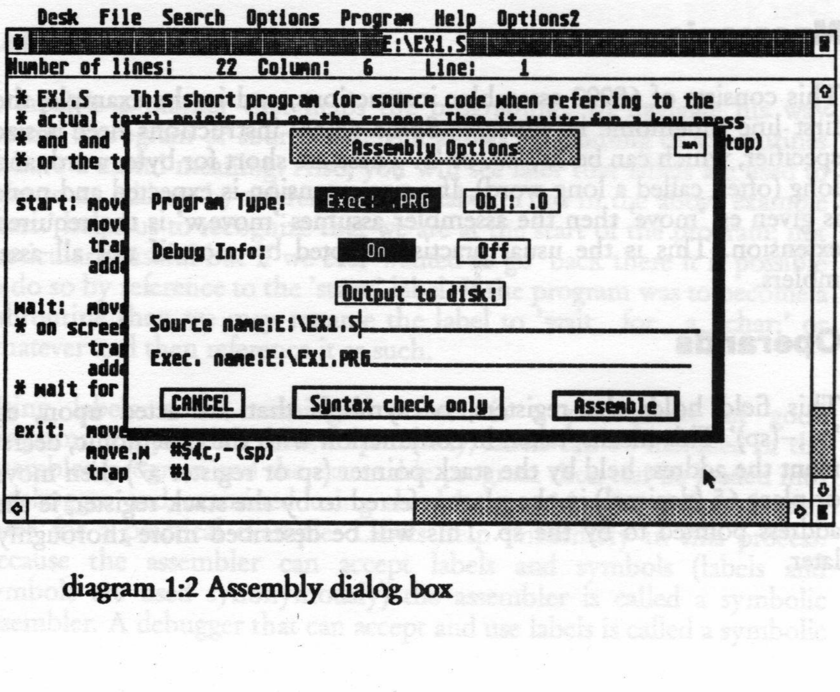

diagram 1:2 Assembly dialog box

If there are any faults in the source code, eg if you typed 'mowe.w' instead of 'move.w' error messages will appear, as the assembler does its work, and tells you on what line(s) the error(s) took place. You will need to correct the error in the text editor before you can assemble and run the program. To do this you should make a note of the line number(s) and then use ALT–G to go to the line number and correct the mistake.

Often source code will assemble correctly (because there were no syntax or other errors), but the program will either not run or will crash or lock up the computer and the only way out is to reset the computer. The problem is usually in the logic of the source code. We may have forgotten some parameter or whatever. This is were the debugger comes into the picture. See later for more details.

Looking at the program line by line

start: move.w #65,-(sp) ; start of program

The label 'start:' just marks the beginning of the program. Only useful if we want to go back to this position. More useful if the program was a subroutine.

As described previously this line of the program decrements the stack pointer (the '–' or minus sign signifies decrement), moves (or places or technically 'pushes') #65, into the place pointed to (or referred to) by the stack pointer address or register a7. But how do I know that this is the required action and why have I done this? To answer the first part of the question we have to look at what we are trying to do, which is to print a character to the screen. Now there is a routine or function in the ROM which does this and ATARI have kindly provided programmers with a method of using this routine (so we don't have to write it ourselves) and a list of all subroutines available to the programmer that are in the ROM is given on the supplied disk or is available in ST technical reference books – see bibliography. This routine is taken from the GEMDOS functions. To access this subroutine we have to pass it the correct parameters via the stack or sp (stack pointer). Why do we have to pass the parameters via the stack? Because that's the way it's done! Passing parameters means to give the subroutine the things we want to give it and the things that it needs to operate correctly. Now this particular subroutine (called 'c_conout' or 'cconout' by the programmers that programmed the ROM) may be expressed in this fashion:

(Note 'c__conout' is probably short for Character CONsole OUT.)

```
move.w    #65,-(sp)
move.w    #2,-(sp)
trap      #1
addq.l    #4,sp
```

End of 'c__conout'subroutine.

Most of the BIOS functions/routines are accessed in this manner. However, when it comes to using GEM ROM based functions we have to use a slightly different technique which is explained later on. Note #65 is the ASCII representation of the character 'A'.

Some assemblers would allow 'move.w 'A',– (sp)' instead of 'move.w #65,– (sp)'.

ASCII stands for American Standard Code for Information Interchange. Alphanumeric characters and other codes, such as those that can be sent to a printer are represented on a computer by their ASCII numbers. A character that is printed to the screen is held in memory in the computer as a bit mapped image, ie each character is made up of little dots which make–up the final image. The way the programmer accesses these bit–mapped characters is via the ASCII table, and is one feature that is common among most makes of computer. ASCII text, can easily be transferred between computers of different operating systems because of its common ground, and between different word processors and text editors. ASCII text is text that is bereft of any command outside the range of common characters of the ASCII table, such as underline, bold. Each word processor uses different methods to signify text attributes such as bold, and underline. Each would use various characters or combination of various characters from the ASCII table to signify different text attributes, so text with these codes needs to be stripped out. It then becomes ASCII text suitable for transfer between various programs and computers. The assembly language programmer soon becomes familiar with the ASCII character codes.

But why '.w' and '.l'? That's what is required too. See chapter two for a discussion about this.

The answer to second part of the question 'why have I done this?' is implicit in the above paragraphs.

move.w #2,-(sp)

This line of the program is similar to line one, and is part of the 'passing parameters' procedure. It tells the o/s that we want to access number TWO routine ie 'c_conout', and

trap #1

that we want to use the GEMDOS. A 'trap' is the name Motorola (manufacturer of the 68000 chip) has given us to tell the o/s we now need to go to the o/s and use a function in the o/s or ROM (o/s and ROM are virtually synonymous). In Z80 assembly language used by such computers as the Spectrum this is known as a CALL to the o/s and the term has been carried over to 16 bit computers as it describes the operation rather well.

addq.l #4,sp

We need to adjust the stack pointer (sp) or register a7 so it it is returned back to its original condition ready for use again. This line of the program does this, by adding four to the 'sp'. The 'q' appended to 'add' stands for quick and can be used if the number is between 0 and 7, inclusive.

wait: move.w #1,-(sp)

The 'wait:' label refers to the part of the program that waits for a key to be pressed. If the three lines of this section were left out then it would be difficult to view the 'A' character on the screen as the program would exit back to the calling program or desktop immediately. Try assembling the program with this section of the program omitted and you will see what is meant. The name given to this is 'conin', probably a shortened version of 'console in'. You might be forgiven if you said 'Well why not call it 'console in'? The reason for shortening the names is that the o/s is set up to recognize only eight characters, and a space is not an allowable character.

trap #1

addq.l #2,sp

These two lines operate as explained previously.

Register 'a7' is used as the stack pointer and should not be used as anything else by the programmer. But what is the stack pointer? The stack pointer is an address that refers to a place in RAM where data may be safely placed for subsequent use by the function called or the programmers use. Data is placed on top of each other in sequential addresses in a manner like stacking plates. However, this means that the last plate (data) has to be removed before the next plate (data) can be accessed. Any address register can be made into a stack pointer but calls to the o/s will expect the address or parameter found in register a7 to be the one it needs.

Before continuing we should look at the way numeric data, and characters are represented to the assembler:

Data representation

For instance the character 'A' can be represented by the equivalent ASCII code of 65, or as 41 in hexadecimal, and 01000001 in binary.

The assembler needs to know exactly what we are referring to otherwise the resultant assembled program will not do what we expect!

Decimal data is prefixed with a hash '#'. Eg #65.

Hexadecimal data is prefixed with a dollar sign $ if it is an address, and #$ if it is a constant, For example #$41 is the same as #65, and although $41 is the same as #65, the assembler would see this as an address in the computer that we were referring to. However, there are exceptions to this rule when defining room for constants or defining space for storage in RAM. This difference will be more apparent later.

Note that in the C language hexadecimal data is written like this:

```
0x10
```

where this equals 16, ie hex data is preceded by '0x'. As much of the ST programming information is written with regard to C it is useful to remember this when studying assembler. Hex data is not acceptable by the assembler in this form.

Binary data is often prefixed with a percentage sign, eg %01000001.

Binary and hex data is often viewed with some apprehension by the beginner, but for programming purposes it is useful and fairly easy to understand.

Hex or hexadecimal data representation is used in computer programming because the basic way data is stored is in bytes, or 8 bits, and grouped in words (2 bytes), and longs (4 bytes). As 4 bits can represent up to 16 different states (0 to 15, or 0 to F in hex) a byte can be represented by a system of numbering with a base of 16. In other words 0 to 9 and A to F are taken to represent these bytes and words. These bits can be either off or on, and usually 1 = on, and 0 = off. These bits are located physically within the RAM of the ST, and a bit that is 'set' or 'on' can signify something as simple as whether a printer is attached.

So, if there are 8 bits to a byte: and these can be represented as %00000000, ie 8 bits that are all off, then %11111111 is all on, how do we transfer that to hex representation. Simply separate the bits into groups of 4 (a nybble would you believe). Binary is known as a base 2 system of counting and is often shown as 111^2, for example.

How do we count in binary? If %0=0 and %1=1 then the way to represent decimal 2 is %10, because if %1 is added to %1, we can't have %2 as this does not exist in binary notation. So we carry 1 and hence get %10. Why not %11, because this =2+1, decimal 3. In the decimal method of counting we get ‹–––1000's 100's 10's units, etc, but in binary we get ‹–––16 8 4 2 1, etc. %1111= decimal 15 or #$F, the maximum that can represented by 4 bits.

So for #$41, we get %0100 0001 separating into nybbles, and this can be figured out this way:

Value of bit if on	8421	8421
	%0100	%0001

Thus we get #$41. Not too difficult. Now let's try converting #$A8 to binary and then to decimal.

Numbers after 9 are represented by A to F up to 15, then a carry is effected. So A=10, b=11 etc. Converting 'A8':

Value of bit if on	8421	8421
Hex	A	8
Binary	1010	1000

Now how do we convert A8 to decimal?:

The simple method is to buy a calculator that does it for you! However, in a similar fashion hex can be treated like binary but as it is base 16 rather than base 2 (binary) we get:

```
Decimal value of hex digit     <---64 32 16 1
Hex                                     A 8
```

To get the correct result we have to multiply each hex digit by its value so the first result is 1*8=8, the second is 10*16=160. Now adding the results together we get 168. Checking with my calculator we get: 168. I'm glad my sums are correct!

Note that the assembler can accept decimal and hex modes of representing data, eg #124, $124, #$124, but cannot accept 'A' as a character string, or binary data, as eg %10000001. Many other assemblers are able to accept all types though.

Signed binary numbers

Often we have to deal with data that is negative, for instance error codes from the BIOS are given as returned negative numbers in register d0. For instance when saving a file to disk if it is write protected then the BIOS will report an error number in register d0. Consider this program fragment which is the code for creating a file.

```
* create_file
    move      #0,-(sp)         ; attribute read/write file
    move.l    #file_name,-(sp) ; address of file name
    move      #$3c,-(sp)       ; create file function number
    trap      #1
    addq.l    #8,sp
    tst       d0
```

```
    bmi         do_error_routine
    move        d0,handle     ; no error so get handle of file

do_error_routine:
* actually exit for the purposes of this short program
    move.w      #20,-(sp) ; leave gracefully!
    move.w      #$4c,-(sp)
    trap        #1

file_name     dc.b    "test.doc",0
handle       ds.w   1
```

Initially a GEM alert box containing the familiar message 'You cannot modify the disk in drive A: because it is write-protected. Before you retry remove write protect.' is presented. Note that this message is shorter on the STE. If cancel is selected then the function falls though.

If register d0 is now examined d0 would contain hex FFFFFFF3, if the disk was write protected, otherwise it would contain a positive number which can be used to access this file at any time until it is 'closed'. See chapters six and seven for more details on files. Other errors could be reported too: disk full, etc. See disk for full list of error codes.

Because negative numbers as well as positive numbers have to be represented in a computer a method called 'two's complement' has been devised so that negative data can be used. The long word FFFFFFFF could be a positive number as hex FFFFFFF3 equals 4,294,967,283 decimal according to my calculator. However the difference between positive and negative numbers is determined by the left-most (most significant) bit of the number, which is known as the sign bit. So FFFFFFF3 is negative as hex F gives all ones in binary notation -1111. If the disk is not write protected then a low positive number, eg 7, can be expected. If it is known that the number returned in d0 will be in 2's complement form then it can be safely assumed that FFFFFFF3 is a negative number. To see this more clearly we need to look at how negative numbers are represented in binary. All the 68000 arithmetic instructions assume signed arithmetic, as do the compare range of instructions.

For instance take the decimal number '1', its binary form (byte) is

0000001

First each bit of the binary number is complemented, ie 0's are replaced with 1's, and 1's are replaced with 0's. So 00000010 becomes

11111110

Then another 1 is always added to give 11111111, which is hex FF, which if extended to fit a long word would give FFFFFFFF, which is –1 decimal. It needs to be sign extended as placing FF in a data register would result in it being accessed in the range –128 to +127 if it was accessed as a byte. To facilitate this action there is a specific 68000 instruction to take care of this 'ext'. To see this clearly please examine the following program fragments.

```
* negative TEST1.S
    clr.l       d0
    move.b      #$ff,d0
    tst.b       d0
    bmi         its_minus
* continue if not -ve number

* negative TEST2.S
    clr.l       d0
    move.b      #$ff,d0
    tst.w       d0
    bmi         its_minus
* continue if not -ve number

* negative TEST3.S
    clr.l       d0
    move.b      #$ff,d0
    ext         d0      ; extend byte FF to word FFFF
    tst.w       d0
    bmi         its_minus
* continue if not -ve number.
```

TEST1.S branches to 'its_minus' as d0 contains #$FF, and as the most significant bit is negative, the number is assumed to be negative.

TEST2.S does not branch to 'its_minus' as a 16 bit number can hold –

32768 to +32767, and in this case FF is seen as 255 decimal.

TEST3.S branches to 'its_minus' as d0 contains #$FFFF, and as the most significant bit is negative, the number is assumed to be negative.

Note the label 'its_minus:' is not shown.

One way to think about FF hex being a negative number is to imagine that you had a milometer that had only 2 digits. It could only count to 99 then it would start again. Adding 1 to 99 would cause the reading to be 0, so relative to 1, 999 can be viewed as a negative number, −1 from 0. In a similar manner the computer carries out arithmetic. When a number gets too big to fit into a byte, word or long, the number wraps around just like in the milometer.

To summarise: Using signed arithmetic a

byte can hold −128 to +127,
a word −32768 to +32767, and a
long word can hold 2,147,483,648 to − 2,147,483,647

What about decimal 2? This is represented in its binary form (byte) as

```
00000010
 11111101    complementing
        1    add 1
---------
 11111110    2's complement
```

This gives FE. Eventually we would get to FFFFFFF3 which is −13 decimal.

HEX	DECIMAL
FF	−1
FE	−2
FD	−3
FC	−4
FB	−5
FA	−6
F9	−7
F8	−8

```
F7    -9
F6    -10
F5    -11
F4    -12
F3    -13
etc
```

Continuing with the analysis of the rest of program:

```
exit:  move.w   #20,-(sp)  ; leave gracefully!
       move.w   #$4c,-(sp)
       trap     #1
* exit from program properly
```

Unless we exit from the program using one of the recommended methods, as the program is exited it will bomb out, ie crash. Not a serious crash (as we have finished with the program), admittedly, but whenever bombs appear it is a signal to the user, and programmer, that something has gone wrong. If it happens in the middle of a program then clearly something is seriously wrong with the program. You could try assembling the source code without the 'exit:' routine. A nice feature of this exit code is that it allows a return code or number to be passed to the calling program. If you use Gribnif's NeoDesk (a replacement desktop shell) you will see that NeoDesk reports that the process exited with a value of 20.

Pterm, for this is the Atari name of the exit process, closes all files (if any where opened), and clears the memory space used by the process or program.

With just a short program a lot of ground has been covered, and many useful points have also been covered, but there is much to look at yet.

Chapter 2
Data Types

In this chapter data lengths (.b, .w, .l), defining data storage and constants (eg ds.b, and dc.b respectively) will be looked at and another simple program will be analysed in some depth.

In the source code examples from the previous chapter, 'move.w' was used quite a lot and 'addq.l' was also used. When using calls to the o/s we are told what length the 'move' and 'addq' operations should use, but as often as not it is up to us to decide. The '.b' suffix stands for byte, '.w' stands for word, and '.l' stands for long or long word:

byte: 8 bits

word: 16 bits (2 bytes)

long: 32 bits (4 bytes or 2 words)

The 68000 has eight data registers, and 8 address registers, a program counter, and a status register. The data registers are referred to as d0 to d7 – 32 bits wide, and the address registers are referred to as a0 to a7 – 32 bits wide too. A register is a held in the actual 68000 processor itself, not in RAM, and as we have already seen by using register a7 they are often used. Data that is held in RAM and ROM is accessed by use of the address registers, as each byte of data has a unique address by which it can be referenced. By manipulating these registers we are able to control the heart of the ST. The fundamental data length used by the 68000 and the ST is the byte, which is 8 bits wide.

One peculiarity of the computing world is that counting always starts at zero (0), so d0 to d7 are eight registers.

The data registers are arranged as overpage:

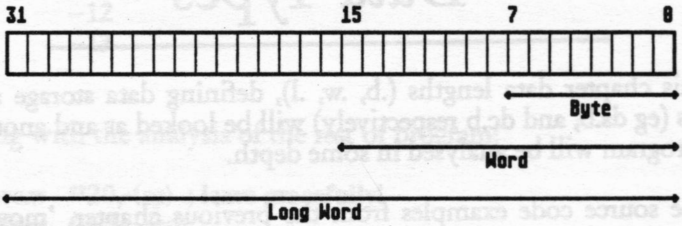

diagram 2:1 Registers

So, if we want to act upon data held in the first 8 bits of a data register we would use the '.b' suffix, and if we wanted to access the first 16 bits we would use the .w suffix, and for all the data length '.l'. Addresses are stored as long words so we usually access addresses using a0–a7 using a '.l', but to access data pointed to by those addresses we can use any of the suffixes, though as previously stated data is held in byte chunks. Enough theory (and if the above is not too clear now it should become clearer as we progress), down to practise!

Let's put our name on the screen!

```
* EX2.S This program prints a string to the screen, waits
* for a key press and exits back.

start:
        move.l   #my_name,-(sp) ; put address of string on stack
        move.w   #9,-(sp)       ; Gemdos function 'print a line', 'Cconws'
        trap     #1
        addq.l   #6,sp          ; correct stack

* wait for key press
        move     #2,-(sp)       ; device number (console)
        move     #2,-(sp)       ; BIOS routine number
        trap     #13           ; Call Bios
        addq.l   #4,sp
```

```
exit:
    move.w   #20,-(sp) ; leave gracefully!
    move.w   #$4c,-(sp)
    trap     #1
* exit from program properly

my_name:     dc.b "Roger Pearson",0
```

This program is slightly different than the first as it uses the 'dc.b' directive. 'dc.b' means define constants in memory of value byte. There can be dc.w, and dc.l as well. So in the above program the label 'my__name:' refers to the address that holds the place were 'Roger Pearson is held. What has happened is a place to hold the string 'Roger Pearson' has been defined at the address 'my__name:'. This address is calculated at assembly time and 'Roger Pearson' is stored there and we can rest assured that the o/s will respect the space allocated. If we looked at address 'my__name:', which we could do as the debugger would allow us to name the label 'my__name:', and the debugger would find the actual address for us. This why it is called a symbolic debugger as it can locate symbols, or labels and, we would see that at the start of that address, the first byte would hold the character 'R', followed by the rest of the string.

Cconws

This has been done as GEMDOS function number nine, Cconws, 'print a line' requires that we pass the address of the string we want printing to the function. The zero (or NULL in computer parlance) after 'Roger Pearson' must be there as GEMDOS sees a null as an end of string marker. If this null was left out everything might be ok, especially if the next byte held in memory was a null, but that cannot be guaranteed so it is always best to append a null after every string. If a null could not be found for a while in RAM then the string might be extremely long and the assembler might be unable to accept it, or if it did the string might be so long as to fill up the screen when run, or it might crash the ST if no null was found. When a '#' is placed before a label then that passes the address of that label, whilst missing out the '#' would pass the contents of label. This will become clearer as we progress. Knowing the difference between passing an address and the contents of that address is crucial to correct programming and will be looked at in more detail later.

The next parameter is the number nine, the Bios function number. Then we 'call' GEMDOS, ie the subroutine ('cconws') is executed, and 'Roger Pearson' is printed to the screen. Then the stack register is corrected as usual.

Bconin

The next three lines which wait for a key press is similar to the three lines in EX1.S, except this time the Bios is used although the effect is the same. This function is called 'Bconin', and it waits for a character from a device, and in this case it waits for a character from the console, but it could equally wait for a character from the other devices as described below. The actual character received is returned in register d0 as a long and in the case of the console the scan code and ASCII code are both returned in d0. The scan code is returned in the least significant byte of the high word, whilst the ASCII code is returned in the least significant byte of the long. The scan code shows what key was struck regardless of whether shift or Alt was pressed. See appendix for list of the key codes returned by Bconin. To see this in actual operation we will use the debugger to inspect register d0, in the next chapter.

ST devices

For the purposes of programming the ST is divided into the following devices. See 'wait for key press' above, as this uses device 2, the console.

Device number	Device name	Description
0	PRN	The parallel port: printer.
1	AUX	The serial port, usually a modem is connected.
2	CON	Keyboard and screen: the CONsole.
3	MIDI	Musical Instrument Digital Interface.
4	IKBD	The intelligent Keyboard Device

Then once again we leave the program and return to the program it was launched from, which would be the text editor if run from there, or the desktop if double-clicked from there. Yes, the desktop is a program run at start–up from the ST's ROM's.

Equates

It is very common to see programs written as EX3.S is. However, EX3.S does the very same as EX2.S except it is written slightly differently – it uses label equates. The label equates given at the start of the program can prove very useful. For instance if a buffer was defined as 'buffer equ 1000', then 1000 (bytes) would be substituted for 'buffer' whenever 'buffer' was used in the program. In a large program we may use 'buffer' many times, and if later a decision to alter the buffer size was made then all that would need changing would be the 'buffer' equate. 'Equ' stands for equals, and other assemblers allow '=' as well.

'Buffer' is a term used a great deal in programming and refers to a given area of memory that has been allocated for whatever purpose the programmer needs it for. The term array is has a similar meaning. It could be said that at the address 'my_name' a buffer the size of the string plus a null is reserved. Note though, that buffers are usually reserved with the directive ds.b, define a space by the size given, eg reserve: ds.b 100 would allocate 100 bytes of free ram from the address 'reserve:'. Similar directives of ds.w, and ds.l also are used. So, we could write reserve: ds.w 50. This will be looked at later in more detail.

```
* EX3.S This program prints a string to the screen, waits for a
* key press and exits back.

gemdos      equ   1
bios        equ   13
cconws      equ   9
pterm       equ   $4c
con         equ   2
bconin      equ   2

start:
      move.l      #my_name,-(sp)  ; put address of string on stack
      move.w      #cconws,-(sp)   ; Gemdos function 'print a line'
      trap        #gemdos
      addq.l      #6,sp           ; correct stack

* wait for key press
```

```
        move     #con,-(sp)    ; device number (console)
        move     #bconin,-(sp) ; BIOS routine number
        trap     #bios         ; Call Bios
        addq.l   #4,sp

exit:
        move.w   #20,-(sp)  ; leave gracefully!
        move.w   #pterm,-(sp)
        trap     #gemdos
* exit from program properly

my_name:        dc.b    "Roger Pearson",0
```

Note that with equates the hex number '4c' is preceded with a '$' rather than a '#$'; this is one case where '$' is not taken to mean an address but an immediate number. Note that when it is used in the program a '#' is placed before 'pterm', and as 'pterm'is defined as '$4c' this changes the value of pterm to #$4c. 'Immediate' data in computerspeak refers to the fact that the number is not an address.

Chapter 3
Looking at the debugger

This chapter takes a quick look at the debugger which is invoked to inspect register d0 after single–stepping the 'bconin' routine as given in EX3.S and continues with another short programming example that looks at the use of the 'ds.b' directive.

One way to use the debugger is to enter it directly after some source code has been assembled correctly. So, if EX3.S is assembled and ALT–D is pressed the debugger will be loaded and the executable file EX3.PRG will be automatically loaded in to the debugger for our inspection.

There is another way to invoke the debugger by pressing ALT–J, or accessing the *Program* drop down menu – do this when you wish to debug another program other than the one just assembled. After you have done this you will be prompted for a filename to pass to the debugger– only executable files may be passed.

Please note that the debugger is taken from the Sozobon PD suite of programs. Thanks to their excellent programming abilities we can now debug our programs with comparative ease.

Debugger commands

The first thing we need to know is what commands the debugger will respond to, and for our purposes they are:

:S	Single step each 68000 instruction, or line of the program and show the registers as well.

:s	As above but do not show the registers.

:C	Execute the program a full speed until a breakpoint is found and show the registers if program is halted through the use of a breakpoint, and shows registers at end of execution.

:c As above except don't show registers.

:b Set a breakpoint.

Control–W See the results of the program in a separate window from the debugger. Pressing the HELP key whilst in the debugger will show some of these commands to the screen.

To explain: single–stepping allows the programmer to execute each line of the program 'step by step' or line by line, and thereby see the results as each line of code is executed.

':C' allows the programmer to run the program until the end if the inspection is over, or to run the program until a breakpoint is set. A breakpoint is positioned in a program so that when the fully running program reaches this point it will stop.

Please note that pressing the Return key will execute the last command given to the debugger.

Unfortunately there are a number of bugs in the debugger. One of them is that it appears to ignore the first line of the program being debugged! It seems to have missed the first line of EX3.PRG 'move.l #my_name,–(sp)'. In fact the debugger has set the program counter, or the position in the program, after the first line of EX3.PRG, but it has also, fortunately, executed the first line too, otherwise the program would not work. In otherwords the address of 'my_name' has been placed on the stack.

So assuming that EX3.S has been assembled correctly and ALT–D has been pressed you should now enter ':S' to single– step through the program. As soon as you have entered ':S' you should press the Return key.

You should see this or something very similar on the screen (see over page):

```
Szadb version 1.0 (english)
> :S
pc        85bcc   xsp        755a   sr 8300 -> User pri3
d0            0   d1            0   d2            0   d3            0
d4            0   d5            0   d6            0   d7            0
a0            0   a1            0   a2            0   a3            0
a4            0   a5            0   a6            0   sp        f3ff4
85bcc:            move.w     #9,-(sp)
>
pc        85bd0   xsp        755a   sr 8300 -> User pri3
d0            0   d1            0   d2            0   d3            0
d4            0   d5            0   d6            0   d7            0
a0            0   a1            0   a2            0   a3            0
a4            0   a5            0   a6            0   sp        f3ff2
85bd0:            trap       #1
> :b
> :C
pc        85bd4   xsp        755a   sr 8300 -> User pri3
d0       fc000c   d1            0   d2            0   d3            0
d4            0   d5            0   d6            0   d7            0
a0        f3ff2   a1            0   a2            0   a3            0
a4            0   a5            0   a6            0   sp        f3ff8
85bd4:            move.w     #2,-(sp)
> :S
```

diagram 3:1

```
> :S
pc        85bd8   xsp        755a   sr 8300 -> User pri3
d0       fc000c   d1            0   d2            0   d3            0
d4            0   d5            0   d6            0   d7            0
a0        f3ff2   a1            0   a2            0   a3            0
a4            0   a5            0   a6            0   sp        f3ff6
85bd8:            move.w     #2,-(sp)
>
pc        85bdc   xsp        755a   sr 8300 -> User pri3
d0       fc000c   d1            0   d2            0   d3            0
d4            0   d5            0   d6            0   d7            0
a0        f3ff2   a1            0   a2            0   a3            0
a4            0   a5            0   a6            0   sp        f3ff4
85bdc:            trap       #d
> :b
> :C
pc        85be0   xsp        755a   sr 8300 -> User pri3
d0     101c000d   d1           f0   d2            0   d3            0
d4            0   d5            0   d6            0   d7            0
a0          c76   a1          93a   a2          c7e   a3            0
a4            0   a5            0   a6            0   sp        f3ff8
85be0:            move.w     #14,-(sp)
> :C
process exited
(hit any key)
```

diagram 3:2

Note 'pc' refers to the program counter, and 'xsp' refers to the supervisor stack pointer, 'sr' refers to the status register. The program counter shows the address of the next instruction the computer will execute.

The ST can run in two states one is called the User State, and the other the Supervisor State. The operating system runs in the supervisor state whilst user programs run in the user state. The supervisor state protects certain areas of memory so that user programs cannot use them, this includes the area of memory that the system variables (see disk for list of system variables) are kept. However, it is possible to access all areas of the ST's memory by going into supervisor mode. This will be shown later.

The status register

The status register (sr) is divided into two equal parts the system byte and the user byte– see diagram. The user byte is what will concern us at a later stage is also known as the 'Condition Code Register' (ccr). Only the low five bits of the user byte are used, and each bit has been given a name, and is used to signify that some state has been reached by a register. To give a brief example, if we wanted to know whether a register contained zero, we could test it using one of the 68000 instructions. When and if the register was zero the zero bit would be set (to one) and would could act upon this information. See chapter six for example of the ccr in use– EX6.S.

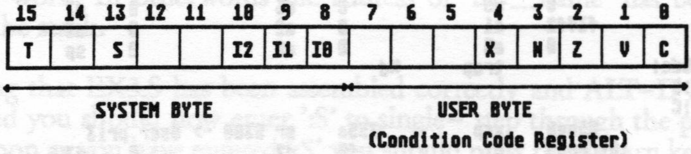

diagram 3:3 The status register

T: 1=Trace mode
S: 1=Supervisor state
I2, O1, I0 Interrupt mask
X: eXtend flag

N:	Negative flag
Z:	Zero flag
V:	oVerflow flag
C:	Carry flag

To continue with using the debugger.

As you can see register d0 contains nothing at this point. Pressing return will execute another single–step of the program and should take us to 'trap #1'. Now single–stepping from this point will take us into the systems ROM or o/s, to execute the routine. It is normal practise to bypass a system call or trap as it is not particularly interesting to go through the ROM, besides it would take a great deal of time! However, feel free, but don't forget to set a breakpoint first as described next! So to get around this problem a breakpoint can be set before the next instruction and after the trap is executed. To do this :b is now entered at the cursor prompt and return is pressed. The breakpoint is now set. As we don't want to single–step each ROM statement we should now enter :C which will run the program at full speed until the breakpoint.

The screen will probably flash (it did for me) indicating the trap has been executed, and now by pressing Control–W you will be able to see my name, 'Roger Pearson', our your name (if you altered the 'my__name' string in the source code), printed on the alternate screen.

Now we are ready to single–step the Bconin routine and inspect register d0.

Another ':S' should be entered and Return pressed, and Return pressed again until 'trap# d' appears. This seems to be another bug, (the 'd') but does not seem to effect the correct operation of the debugger. What should have been printed is' trap #1'. However, if we continue by entering ':b', followed by ':C' then the program will display the alternative screen with 'my__name' string displayed. The program is waiting for a key press. At this stage I pressed the Return key, and register d0 contained '101c000d'. If you look at the appendix Key Codes you will see that the key code for Return is '1c0d'. '1c' being the actual scan code and '0d' being the ASCII code of the Return key.

Next another ':C' should be entered and then the program will run until the end, and then the debugger will exit back to the text editor after any key is pressed, by following the debugger command '(hit any key)'.

Using'globl'

As stated earlier the debugger is symbolic, ie it can load and use the labels or symbols that we have used in our source code. To access the labels we use a '.globl' command which has to placed before any label that we want the debugger to use. The '.globl' command tells the assembler that labels should be kept in the executable file. Normally the executable file does not contain any labels unless '.globl' is used, only their locatable addresses generated by the use of labels in the source code. Labels are only dumped (to use the correct jargon) in the executable file when we want to debug it. Labels are invariably not dumped when the final executableis generated, as labels add to the length of the executable code and can aid others to debug/steal/understand your code much easier. If you can load your executable file in a debugger so can others! If a table of all labels are generated by the debugger or some times a separate program it is known as a symboltable.

'EX3A.S' shows how to use the '.globl' command, and as the first line bug would mean that we could not see it another line of code has been entered prior to the use of the 'my__name' label, 'move.l#1,d1'. This extra code does not affect the program in anyway. If 'move.l' was inserted into EX3.S you would only see an address rather than the 'my__name' label.

Note that it is not possible to see comments in a debugger, as comments are not assembled at any time to an executable file.

```
* EX3A.S This program prints a string to the screen, waits for a
* key press and exits back.

gemdos          equ     1
bios            equ     13
cconws          equ     9
pterm           equ     $4c
con             equ     2
bconin          equ     2

start:  move.l  #1,d1           ; only here for debugger bug!
        move.l  #my_name,-(sp)  ; put address of string on stack
```

```
        move.w      #cconws,-(sp)  ; Gemdos function 'print a line'
        trap        #gemdos
        addq.l      #6,sp          ; correct stack

* wait for key press
        move        #con,-(sp)     ; device number (console)
        move        #bconin,-(sp)  ; BIOS routine number
        trap        #bios          ; Call Bios
        addq.l      #4,sp

exit:
        move.w      #20,-(sp)  ; leave gracefully!
        move.w      #pterm,-(sp)
        trap        #gemdos
* exit from program properly

            .globl my_name
my_name:    dc.b    "Roger Pearson",0
```

The ds (define space) directive.

The 'ds' directive is similar to the 'dc' directive that was looked at earlier, and can be used in three different ways: ds.b, ds.w, and ds.l.

So, 'ds.b' means reserve a enough space in RAM for a byte of data. Similarly, 'ds.w' means allocate room in RAM for a word of data, whilst 'ds.l' reserves a long word of data (or a 'long' as it's often termed). Note that data reserved using the 'ds' directive is initialised to zero until used by the programmer. Data is reserved in our source code similarly to the method used for the 'dc' directive:

```
label  ds.b   4    ; reserve 4 bytes at address label
```

note that

```
label ds.l   1     is exactly the same.
```

Why should we want to reserve space in RAM? There are many reasons why we should want to do this, eg if we had something on the screen that we wanted to keep whilst we loaded a DEGAS screen from disk to display on the screen we would have to reserve 32000 bytes (32K) to store the screen whilst it was occupied by the DEGAS picture. The ST's screen uses 32000 bytes of RAM whether in low, medium, or high resolution. For a more detailed description of the ST's screen see chapter six and eight. To reserve 32K or 32000 bytes we would write:

```
save_screen    ds.b    32000.
```

If we wanted to keep a screen in RAM for whatever reason then all we would need to do would be to copy the screen starting at address 'save__screen', which in assembler is quite easy. See this chapter for demonstration of using a 32K buffer.

The next chapter shows the use of the define space ('ds') directive and the use of a subroutine.

Chapter 4
'ds' and 'jsr'

In this chapter the define storage (ds) directive and the jump to subroutine 'jsr' instruction are examined. EX4.S is a practical example.

```
* EX4.S
* This program finds the address of the screen, prints 'my_name'
* string to screen, clears the screen, and exits back.

gemdos equ      1
bios    equ     13
xbios   equ     14
cconws equ      9
pterm   equ     $4c
con     equ     2

start:
        move    #2,-(sp)            ; get screen RAM address
*(physical base), returned in d0
        trap    #14                 ; call Xbios
        addq.l  #2,sp               ;  correct stack
        move.l  d0,screen_address   ; put screen address in symbol

        move.l  #my_name,-(sp) ; put address of string on stack
        move.w  #cconws,-(sp)  ; Gemdos function 'print a line'
        trap    #gemdos
        addq.l  #6,sp          ; correct stack

* this goes to the address 'wait_for_key_press' and executes the
* short routine held there until an rts' is found.

        jsr     wait_for_key_press

* lets clear the screen
        move.l  #31999,d0           ; counter #32000-1
        move.l  screen_address,a0   ; place screen address in an
* address register
```

```
do_it_again:
        clr.b   (a0)+                   ; now clear the screen
        dbra d0,do_it_again

* wait for a key press so that we can see that the screen has been
* cleared before exiting to desktop or text editor

        jsr     wait_for_key_press

exit:
* exit from program properly
        move.w  #20,-(sp)  ; leave gracefully!
        move.w  #pterm,-(sp)
        trap    #gemdos

************* subroutine *****************
wait_for_key_press:
* wait for key press subroutine
        move    #con,-(sp)      ; device number (console)
        move    #2,-(sp)        ; BIOS routine number
        trap    #bios           ; Call Bios
        addq.l  #4,sp
        rts

my_name:        dc.b    "Roger Pearson",0
screen_address:         ds.l    1
```

Examining EX4.S we can see there is a number of features that need some explanation.

First the address of the screen is found, and to do this the Xbios function number 2 is used. This function finds the address of the screen and returns it in register d0. In some computer systems the screen is always held at one particular address, but the ST's screen can be placed anywhere, and it differs especially between 520's, and 1040's, at boot up. As we are not going to use the screen address just yet it is placed in the symbol 'screen_address' until it is needed. Why not just leave it in register d0? The reason is that calls to the o/s whether the BIOS or XBIOS often use registers a0–a2, d0–d2 to return parameters asked for, or use the registers themselves as we have seen previously with Bconin.

So it is best to store data that we need in a safe place until it is needed.

Next the 'cconws' function is called and the 'my__name' string is printed to the screen.

Next is:

jsr wait_for_key_press

'jsr' means 'jump to subroutine' at address 'wait__for__key__press' and execute whatever is there until an 'rts' is found. 'rts' means 'return from subroutine'. In this case the subroutine is the familiar 'wait for a key press'.

Quite often, and it is safer to do so, all the registers are stored (or 'pushed') onto the stack prior to entering a subroutine, and at the end of the subroutine all the registers are taken or 'popped' from the stack. This ensures that whatever is contained in any register remains unaffected by the subroutine's action. Subroutines can be many lines long and many data and address registers may be used in the course of its action.

So, taking the 'wait__for__key__press' subroutine we would get this:

```
wait_for_key_press:
        movem.l a0-a6/d0-d7,-(sp)
        move    #con,-(sp)  ; device number (console)
        move    #2,-(sp)  ; BIOS routine number
        trap    #bios   ; Call Bios
        addq.l  #4,sp
        movem.l (sp)+,a0-a6/d0-d7
        rts
```

'movem' mnemonic means move multiple registers, and any combination of address and data registers can be stored, eg

movem.l a0/d0-d2,-(sp)

Note that it is usual to use 'long' when storing registers onto the stack as this means that nothing is left to chance as all the data in the various registers are retrieved to safety. Note though that any value returned in register d0 within the subroutine cannot be examined after returning from the subroutine as it will be lost when the registers are returned to their original values just before the 'rts'.

If we needed to use the data in register d0, it would be necessary to set up some storage space at a label.

```
my_data      ds.l    1
```

and then use this storage space to store the data:

```
wait_for_key_press:
        movem.l a0-a6/d0-d7,-(sp)
        move    #con,-(sp)      ; device number (console)
        move    #2,-(sp)        ; BIOS routine number
        trap    #bios           ; Call Bios
        addq.l  #4,sp
        move.l  d0,my_data
        movem.l (sp)+,a0-a6/d0-d7
        rts
```

Now we come to the clearing the screen part of the program:

```
* lets clear the screen
        move.l  #31999,d0               ; counter #32000-1
        move.l  screen_address,a0       ; place screen address in an
* address register

do_it_again:
        clr.b   (a0)+                   ; now clear the screen
        dbra    d0,do_it_again          ; dbra = decrement and branch

*until false
```

First the number of bytes, minus one is placed in d0. We subtract one because of the 'dbra' mnemonic at the end of the clear screen routine, as 'dbra' means decrement a register until the value of the register is false, or less than zero (the 'bra' means 'branch always', 'dbra' means decrement and branch always until false). If the condition is not false then it branches or jumps to the address given in the operand field, in this case 'do_it_again'. If one is not subtracted from 32000 then the loop would operate 32001 times.

'clr.b' means clear a byte (set all bits to zero), and here the byte is the one held at the screen's address. '(a0)+' means get the contents of the address (the parenthesis indicate this), and after the 'clr' instruction has operated on this byte of memory increment the memory address so that the next byte can be accessed. This method of addressing data is called 'indirect with post–increment'. Other addressing modes will be looked at in chapter four.

The ST's screen occupies 32K of RAM, and each individual bit of memory signifies the status of each specific point or pixel (PICture ELements – the individual dots that make up a monitor or TV screen). If a bit is set to one the colour of this pixel will be black – at least on a monochrome TV or monitor. Each pixel is assigned a colour in medium resolution and low resolution, though the effect of clearing each byte of memory is the same. When a bit is cleared then that pixel turns white, and clearing 32K bytes results in the whole screen being cleared of any information.

So what we have in this routine is a very fast loop which clears all the individual bits that make up the screen. If you run this program you will be able to see how fast it operates. However there are other ways to write the 'do_it_again' routine:

```
* lets clear the screen
        move.l  #7999,d0                ; counter #32000/4 -1
        move.l  screen_address,a0       ; place screen address in an
* address register

do_it_again:
        clr.l   (a0)+                   ; now clear the screen
        dbra    d0,do_it_again
```

This time instead of a byte being cleared a long is cleared. A long is equivalent to four bytes so if the counter is divided by four and then decremented by one because of the 'dbra' then the counter will be the correct value.

However, the fastest way to clear the screen is probably the method shown below as more time is spent clearing the screen rather than looping. However there is a trade–off here as the source code is longer, which can be important.

```
* lets clear the screen
        move.l   #1999,d0                ; counter #32000/4/4 -1
        move.l   screen_address,a0       ; place screen address in an
* address register

do_it_again:
        clr.l    (a0)+                   ; now clear the screen
        clr.l    (a0)+
        clr.l    (a0)+
        clr.l    (a0)+
        dbra     d0,do_it_again
```

This time as we are using the 'clr.l' instruction four times in the loop then we need to divide 32000 once again.

It would be usual to have the 'clear the screen' routine as a subroutine in any program of length as clearing the screen is often used. It would be set up in the same way as 'wait__for__key__press' subroutine was using the 'movem.l' instruction. Viz:

```
clear_the_screen
        movem.l a0-a6/d0-d7,-(sp)
* lets clear the screen
        move.l   #1999,d0                ; counter #32000/4/4 -1
        move.l   screen_address,a0       ; place screen address in an
* address register

do_it_again:
        clr.l    (a0)+                   ; now clear the screen
        clr.l    (a0)+
        clr.l    (a0)+
        clr.l    (a0)+
        dbra     d0,do_it_again
        movem.l (sp)+,a0-a6/d0-d7
        rts
```

By storing the screen in a buffer 32K long then it is possible to rewrite the buffer back to the screen so that any information that was on the screen initially will be replaced. This is one use of an 'UNDO' key in a lot of programs, eg in an art program we may have done something we did not like so we press the UNDO key and the screen is restored to the condition it was prior to the mistake. Obviously it would depend on when the screen was saved as to what the replaced screen would be like.

Anyway, the source code could look like this:

```
* EX5.S
* This program finds the address of the screen, prints 'my_name'
* string to screen, clears the screen, and prints the string 'my_name'
* again.

gemdos equ    1
bios   equ    13
xbios  equ    14
cconws equu   9
pterm  equ    $4c
con    equ    2

start:
       move     #2,-(sp)               ; get screen RAM address
* returned in d0
       trap     #xbios                 ; call Xbios
       addq.l   #2,sp                  ; correct stack
       move.l   d0,screen_address      ; put screen address in symbol

       move.l   #my_name,-(sp)  ; put address of string on stack
       move.w   #cconws,-(sp)   ; Gemdos function 'print a line'
       trap     #gemdos
       addq.l   #6,sp           ; correct stack

* save the screen in a buffer

save_the_screen:
       move.l   #1999,d0               ; counter #32000/4/4 -1
       move.l   screen_address,a0      ; place screen address in an
* address register
       move.l   #screen_buffer,a1      ; address of screen buffer in
```

```
* a1
save_it_again:
        move.l  (a0)+,(a1)+             ; save the screen in
* screen_buffer
        move.l  (a0)+,(a1)+
        move.l  (a0)+,(a1)+
        move.l  (a0)+,(a1)+
        dbra    d0,save_it_again

        jsr     wait_for_key_press

        jsr     clear_the_screen

        jsr     wait_for_key_press

buffer_to_screen:
        move.l  #1999,d0                ; counter #32000/4/4 -1
        move.l  screen_address,a0       ; place screen address in an
* address register
        move.l  #screen_buffer,a1       ; address of screen buffer in
* a1
put_it_again:
        move.l  (a1)+,(a0)+ ; place contents of 'screen buffer'
* to screen
        move.l  (a1)+,(a0)+
        move.l  (a1)+,(a0)+
        move.l  (a1)+,(a0)+
        dbra    d0,put_it_again

        jsr     wait_for_key_press

exit:
* exit from program properly
        move.w  #20,-(sp)   ; leave gracefully!
        move.w  #pterm,-(sp)
        trap    #gemdos

************** subroutines ******************
wait_for_key_press:
* wait for key press subroutine
        move    #con,-(sp)      ; device number (console)
```

```
        move      #2,-(sp)            ; BIOS routine number
        trap      #bios               ; Call Bios
        addq.l    #4,sp
        rts

clear_the_screen:
        movem.l a0-a6/d0-d7,-(sp)
* lets clear the screen
        move.l    #1999,d0                      ; counter #32000/4/4 -1
        move.l    screen_address,a0             ; place screen address in an
* address register

do_it_again:
        clr.l     (a0)+                         ; now clear the screen
        clr.l     (a0)+
        clr.l     (a0)+
        clr.l     (a0)+
        dbra      d0,do_it_again
        movem.l (sp)+,a0-a6/d0-d7
        rts

my_name:          dc.b     "Roger Pearson",0
screen_address:     ds.l     1
screen_buffer:  ds.b     32000
```

Now the executable file for this program is not on the disk! This is because once assembled the executable file is over 32K in length, obviously due to the 32K screen buffer. Have a look for yourself if you have just assembled 'EX5.S'. Fortunately there is an easy way around this, as there is no real point in saving 32K of empty space to disk, and that is to specify that we want the screen buffer to be held in the 'bss', or block storage segment. This means that the 32k buffer is not saved to disk only the information that we want a 32k buffer after the program has been loaded. The code for this is to place a 'bss' before the define storage directives that need to be specified for the 'bss'. The 'bss' stores unitialised data, ie data that may not necessarily be empty of data. Initialised data is always set to zero, so we can be sure that the space really is empty. However, in EX5.S data is placed into the buffer over writing any data in it so it does not need to be initialised to zero first.

```
my_name:   dc.b   "Roger Pearson",0
screen_address:   ds.l   1

        .bss
screen_buffer:   ds.b   32000
```

When the source code with the altered '.bss screen__buffer:' is assembled the resultant executable file is only 214 bytes in size! What a difference. This the executable file on the zzSoft disk.

If you had a software company you could soon impress your prospective customers with the size of your executable programs by making sure that you never used the bss!

Note that '.bss' or 'bss' are both acceptable to the assembler.

Another method of placing an address on the stack is to use the 'pea' instruction, Push Effective Address. EX4A.S shows how 'pea' is used.

```
* EX4A.S
* This program demonstrates the use of the 68000 'pea' instruction.

gemdos equ    1
bios   equ    13
cconws equ    9
pterm  equ    $4c
con    equ    2

start:
       pea      my_name       ; put address of string on stack using
* pea
       move.w   #cconws,-(sp) ; Gemdos function 'print a line'
       trap     #gemdos
       addq.l   #6,sp         ; correct stack

* wait for key press
       move     #con,-(sp)    ; device number (console)
       move     #2,-(sp)      ; BIOS routine number
       trap     #bios         ; Call Bios
       addq.l   #4,sp
```

```
exit:
        move.w   #20,-(sp)  ; leave gracefully!
        move.w   #pterm,-(sp)
        trap     #gemdos
* exit from program properly

my_name:        dc.b     "Roger Pearson",0
```

```
my_name:   dc.b     "Roger Pearson",0
screen_address:  ds.l     1

         .bss
screen_buffer:  ds.b     32000
```

When the source code with the altered 'bss-screen_buffer:' is assembled the resultant executable file is only 214 bytes in size! What a difference. This the executable file on the zzSoft disk.

If you had a software company you could soon impress your prospective customers with the size of your executable programs by making sure that you never used the bss!

Note that '.bss' or 'bss' are both acceptable to the assembler.

Another method of placing an address on the stack is to use the 'pea' instruction, Push Effective Address. EX4A.S shows how 'pea' is used.

```
* EX4A.S
* This program demonstrates the use of the 68000 'pea' instruction.

gemdos equ     1
bios   equ     13
cconws equ     9
pterm  equ     $4c
con    equ     2

start:
       pea      my_name       ; put address of string on stack using
* pea
       move.w   #cconws,-(sp)   ; Gemdos function 'print a line'
       trap     #gemdos
       addq.l   #6,sp           ; correct stack

* wait for key press
       move     #con,-(sp)      ; device number (console)
       move     #2,-(sp)        ; BIOS routine number
       trap     #bios           ; Call Bios
       addq.l   #4,sp
```

Chapter 5
Addressing Modes

Addressing modes were briefly mentioned in chapter three, but as addressing modes are part and parcel of the study of assembly language here is a description of them. Most of this chapter can be used for reference as and when you need information on a particular addressing mode – no point in getting a headache just yet!

The 68000 has a total of 14 addressing modes.

The notation 'Dn', where 'n' is a register number from 0 to 7 is often used in as shorthand to describe the data registers, similarly 'An' for address registers 0 to 7. Eg, MOVE.L (An)+,Dn

1. Inherent addressing

In this addressing mode there are no operands since they are already supplied by the opcode. For example,

Reset

Reset is an 68000 instruction which is used to reset all the peripherals.

2. Data register direct

This mode specifies that the operand should be found in one of the data registers. For example move the contents of data register d1 to data register d0:

Instruction	Before	After
MOVE.B D1,D0	d0=ffffffff d1=01234567	d0=ffffff67 d1=01234567
MOVE.W D1,D0	d0=ffffffff d1=01234567	d0=ffff4567 d1=01234567
MOVE.L D1,D0	d0=ffffffff d1=01234567	d0=01234567 d1=01234567

An instruction with .b as a suffix only changes the lowest 8 bits of the destination, and instructions with .w as a suffix only change the lowest 16 bits of the destination. Instructions with .l as a suffix change all 32 bits of the destination.

3. Address register direct

In this addressing mode an address register should be one of the operands. Byte operators (those with .b suffix) are not allowed in this addressing mode. When using the address register as a destination and a word operation (suffix .w) is used, the destination word is sign–extended into a long word. This means that during a word transfer into an address register the upper 16 bits are filled with the value of the most–significant bit (this is bit 15) of the word. The example below will show you how it's done.

Instruction	**Before**	**After**
MOVE.W A1,D0	d0=ffffffff a1=01234567	d0=ffff4567 a1=01234567
MOVE.W D0,A1	d0=01234567 a1=ffffffff	d0=01234567 a1=00004567 <- extended
MOVE.W D0,A1	d0=0000ffff a1=00000000	d0=0000ffff a1=ffffffff <- extended
MOVE.L A1,D0	d0=ffffffff a1=01234567	d0=01234567 a1=01234567

4. Address register indirect

In this addressing mode, the address register contains the address of the memory location that points to contents of that address. In assembler this is being denoted by putting parentheses around an address registers name, e.g. (a0).

When using word '.w' or longword '.l' addressing it is necessary that the address register contains an even number.

Instruction	Before	After
MOVE.L (A1),D0	d0=ffffffff a1=00005000 address $5000 contains 01234567	d0=01234567 a1=00005000
MOVE.L D0,(A1)	d0=76543210 a1=00005000 address $5000 now contains 76543210	d0=76543210 a1=00005000

5. Address register indirect with post-increment

This addressing mode resembles the address register indirect addressing mode. The only difference is that after having moved or stored the data, the address register is incremented. The amount incremented depends on the suffix used in the opcode. If the suffix is .b then the address register will be incremented by one. If the suffix is .w then the address register will be incremented by two (one word is two bytes). If the suffix is .l then the address register will be incremented by four (one longword is four bytes). In assembler this addressing mode is denoted by putting the address register within parentheses followed by a + sign. For example: (a0)+.

Instruction	Before	After
MOVE.L (A1)+,D0	d0=ffffffff a1=00005000	d0=01234567 a1=00005004 <– in-
cremented by 4	address $5000 contains 01234567	
MOVE.W (A1)+,D0	d0=ffffffff a1=00005000	d0=ffff0123 a1=00005002 <– in-
cremented by 2	address $5000 contains 01234567	
MOVE.B (A1)+,D0	d0=ffffffff a1=00005000	d0=ffffff01 a1=00005001 <– in-
cremented by 1	address $5000 contains 01234567	
MOVE.L D0,(A1)+	d0=76543210	d0=76543210

a1=00005000 a1=00005004
address $5000 now contains 76543210

For instance to search for a character sting until the terminating null character is found can be implemented like this. Assuming the address of the string is in address register A1. Note that a NULL is used by GEM as an end of string marker.

```
loop:    tst.b    (a1)+    ; test to see if there a null. Flag
* in CCR set to1 if null found. If not then
         bnz      loop     ; branch if not zero
```

6. Address register indirect with pre-decrement

This addressing mode resembles the address register indirect addressing mode. The only difference is that before moving or storing the data, the address register is decremented. The decrement depends on the suffix used in the opcode. If the suffix is .b then the address register will be decremented by one. If the suffix is .w then the address register will be decremented by two (one word is two bytes). If the suffix is .l then the address register will be decremented by four (one longword is four bytes). In assembler this addressing mode is denoted by putting the address register within parentheses preceded by a – sign. For example: –(a0)

Instruction	Before	After
MOVE.L –(A1),D0	d0=ffffffff d1=00005004	d0=01234567 a1=00005000 <– decremented by 4
	address $5000 contains 01234567	
MOVE.W –(A1),D0	d0=ffffffff a1=00005004	d0=ffff4567 a1=00005002 <– decremented by 2
	address $5000 contains 01234567	
MOVE.B –(A1),D0	d0=ffffffff a1=00005004	d0=ffffff67 a1=00005003 <– decremented by 1
	address $5000 contains 01234567	

MOVE.L D0,-(A1)	d0=76543210 a1=00005004	d0=76543210 a1=00005000
	address $5000 now contains 76543210	

7. Address register indirect adressing with displacement

Assembler syntax: w(An) (w stands for word displacement)

This addressing is also rather similar to address register indirect addressing. The only difference lies in the fact that before moving or storing the data a 16-bit signed displacement is added to the contents of the address register (the address register itself does not change). In assembler this addressing mode is denoted by enclosing the address register name in parentheses preceded by a 16-bit constant. For example: 8(a6) denotes the memory location whose address is the contents of a6 plus 8.

This addressing mode is very useful for accessing data structures. Note if a $ is placed before a number then the data is taken to be hex, otherwise decimal data is assumed by the assembler.

Instruction	Before	After
MOVE.L 8(A1),D0	d0=ffffffff a1=00005000	d0=01234567 a1=00005000
	address $5008 contains 01234567	
MOVE.L D0,-6(A1)	d0=76543210 a1=00005006	d0=76543210 a1=00005006
	address $5000 now contains 76543210	

8. Address register indirect with index

Assembler syntax: b(An,Rn.w) or b(An,Rn.l)

(R stands for a register).

This addressing mode makes it possible to add a variable index (contained in an address or data register) to an address register and also an eight bit signed displacement. The variable index may be either word or longword. Both the index and displacement are sign extended before they are added to the address register.

Instruction	Before	After
MOVE.L 8(A1,A0.L),D0	d0=ffffffff	d0=01234567
	a1=00001000	a1=00001000
	a0=00078000	a0=00078000
	address $79008 contains 01234567	
MOVE.L 8(A1,A0.W),D0	d0=ffffffff	d0=01234567
	a1=00001000	a1=00001000
	a0=00078000	a0=00078000
note a0.w=8000 -> sign-extend gives ffff8000		
	address $ffff8008 contains 01234567	
MOVE.W 8(A1,D0.L),D0	d0=0001fffe	d0=00010123
	a1=00001000	a1=00001000

```
00001000 (contents of a1)
0001fffe (contents of d0.l)
00000008 (sign-extended byte displacement)
----------
00021006
```

address $21006 contains 01234567

Instruction	Before	After
MOVE.L 8(A1,D0.W),D0	d0=0001fffe	d0=01234567
	a1=00001000	a1=00001000

```
00001000 (contents of a1)
fffffffe (sign-extended contents of d0.w)
00000008 (sign-extended byte displacement)
----------
00001006
```

address $1006 contains 01234567

9. Absolute short addressing

With absolute short addressing it is only possible to specify a 16 bit constant. At execution time the 68000 sign extends the word into a long address.

Instruction	Before	After
MOVE.L $1234,D0	d0=ffffffff	d0=01234567
	address 1234 contains 01234567	
MOVE.L $5000,D0	d0=ffffffff	d0=76543210
	address $ffff5000 contains 76543210	

10. Absolute long addressing

With this addressing mode a long address is supplied. It is very similar to absolute short addressing.

Instruction	Before	After
MOVE.L $12345678,D0	d0=ffffffff	d0=01234567
	address $00345678 contains 01234567	

11. Program counter with displacement

Assembler syntax: x(PC) (x is a 16 bit constant)

This addressing mode is the same as address register indirect with displacement. The only difference is that the address register is replaced with the PC (the PC is in fact also an address register).

Instruction	Before	After
MOVE.L 8(PC),D0	d0=ffffffff	d0=01234567
	pc=00005000	pc=00005000
	address $5008 contains 01234567	

12. Program counter with index

Assembler syntax: b(PC,Rn.L) or b(PC,Rn.w) (b is 8 bits)

This mode is the same as address register indirect addressing with index.

Instruction	Before	After

MOVE.L 8(PC,A0.L),D0	d0=ffffffff pc=00001000 a0=00078000 address $79008 contains 01234567	d0=01234567 pc=00001000 a0=00078000
MOVE.L 8(PC,A0.W),D0	d0=ffffffff pc=00001000 a0=00078000	d0=01234567 pc=00001000 a0=00078000

Note: a0.w=8000 –> sign–extend gives ffff8000

address $ffff8008 contains 01234567

MOVE.W 8(PC,D0.L),D0	d0=0001fffe pc=00001000	d0=00010123 pc=00001000

```
00001000 (contents of pc)
0001fffe (contents of d0.l)
00000008 (sign–extended byte displacement)
----------
00021006
```

address $21006 contains 01234567

MOVE.L 8(PC,D0.W),D0	d0=0001fffe pc=00001000	d0=01234567 pc=00001000

```
00001000 (contents of pc)
fffffffe (sign–extended contents of d0.w)
00000008 (sign–extended byte displacement)
----------
00001006
```

address $1006 contains 01234567

13. Immediate addressing

Instruction	Before	After
MOVE.L #$A1234E5D,D0	d0=00000000	d0=A1234E5D

14. Status condition code register addressing

Assembler syntax: SR or CCR

SR = Status Register. CCR = Condition Code Register

This mode is used to control the contents of this register. Changes to the SR can only be made when in user-mode. Changes to the CCR can be made in any mode. Note 'SR'and 'CCR' are reserved words in the assembler, ie don't use them as labels.

Instruction	**Before**	**After**
MOVE.W SR,D0	d0=87654321 sr=3200	d0=87653200 sr=3200
MOVE.W #$0500,SR	sr=3200	sr=0500

Notice that the 68000 was in supervisor mode before executing the instruction but after completion it is in user mode. This operation isn't possible the other way around.

A summary of the address modes of the 68000.

Syntax	**Name**
Dn	Data register direct
An	Address register direct
(An)	Address register indirect
(An)+	Address register indirect with post-increment
-(An)	Address register indirect with pre-decrement
w(An)	Address register with displacement
b(An,Rn)	Address register with index
w	Absolute short
l	Absolute long
w(PC)	PC with displacement
b(PC,Rn)	PC with index
#x	Immediate
SR or CCR	Status register

b is a byte constant
w is a word constant
l is a long constant
x any of b, l or w

n is a register number ranging from 0 to 7
R is a register specifier, either A or D

Chapter 6
Files and the Screen

This chapter looks at how to load files from and to floppy disk, and an example program loads a DEGAS file and displays it on the screen. The format of screen RAM is looked at.

Loading a file from disk is often referred to as 'reading', and saving a file to disk is often referred to as 'writing'.

Reading a file from disk:

There are three stages to loading a file:

(1) Opening the file
(2) Actually reading the file into a buffer.
(3) Closing the file

(1) First a file must be opened. GEMDOS allows 40 files to be open at any one time and to distinguish between them each opened file is given a number from which it can always be identified. Even if only one file is opened a number is always allocated to that file. The number allocated to it is called its 'handle'.

The source code for opening a file is:

```
* open a file
    move.w   #0,-(sp)          ; set file attribute
    move.l   #file_name,-(sp)   ; address of filename
    move.w   #$3d,-(sp)        ; open function number
    trap     #1                ; hello GEMDOS
    add.l    #8,sp
    tst      d0                ; -ve number?
    bmi      error_routine      ; yes, go to error routine
    move.w   d0,handle          ; store file handle

error_routine:
......................
```

....

```
handle:         ds.w    1
file_name:      dc.b    'XX.PI3',0
```

The source code is not complete as the routine 'error routine' is not included, but it serves the purpose of introducing stage one of loading a file.

A '.PI3' file is an uncompressed DEGAS ELITE file. DEGAS was the one of the first art package released for the ST and as such its file format became the standard used for picture files from then on. '.PI3' refers to a high res file, whereas '.PI2' is a medium resolution file, and '.PI1' refers to a low res PIcture file. A compressed DEGAS PIcture file changes the 'I' for a 'C', so that a high res compressed DEGAS ELITE file has the extension '.PC3'.

To open a file two parameters need to be passed to it – the file name, and the file's attribute. The file's attribute is set when the file is Saved, and is usually 0, for normal 'read or write'. The different file attributes are given below:

$00 Normal file, can read and write to it.
$01 File is read only (can't be deleted or written to)
$02 File is on disk, but does not appear in file selector directory, ie hidden
$04 System file
$08 File is volume name, ie disk name
$10 File is subdirectory/folder

Normally when trying to load a file a GEM selector box, or more accurately a GEM dialog box is used. A directory listing is made of all the files that fulfil the pattern described in the mask on the command line. For instance when a directory listing of all files with the extension .DOC are needed, then '*.DOC' is used. '*' is a shorthand way of saying 'anything', so '*.*' would list all of the files on the disk. Using a file selector box to load and save files is looked at later on.

If the file exists then a non-negative number is returned in register d0, which is the file's 'handle', and is used from now on when ever we refer to the file. If a negative number is returned in d0, the 'tst' instruction sets the flags in the condition code register (ccr). Next the 'bmi' instruction is used to alter the course of the program if the 'tst' instruction actually set the ccr negative flag to 1, (set actually means 'set to one', 'reset' equals 'set to zero'). 'bmi' means 'branch if minus' (minus =negative), and in our case means branch to 'error_routine'. 'error_routine' would then display a 'file not found' message, possibly with various options, for instance to try and find the file again. An example of this will be included in the final source code example at the end of this chapter.

In the debugger all 16 bits of the Status Register (sr) are shown, for example, when I used the debugger to examine EX6.S the sr showed the following when 'tst' was single stepped:

before
  sr 8300 –› User pri3
after
  sr 8308 –› User pri3 N

showing that the negative flag had been set. The last '8' in the 8308 shows bit 3 (counting from 0) has been set, and the debugger displays an 'N' to also remind us that the negative flag has been set.

The GEMDOS error codes returned in register d0 when file loading or saving is not successful are as follows:

- –32 Invalid function number
- –33 File not found
- –34 Path not found (see explanation below of 'path')
- –35 Too many open files (no handles left)
- –36 Access not possible
- –37 Invalid handle number
- –39 Not enough memory
- –40 Invalid memory block address
- –46 Invalid drive specification. Ie drive does not exist.
- –49 No more files (used when searching directories/folders)

'path' refers to the specification that is given whenever a file is searched for. For instance folders (or directories) are often used to collect together a certain type of file, so that for instance all DEGAS picture files may be placed in a folder called 'PICTURES' on a disk in drive 'B'. The way a '.PI3' file would be accessed would be as 'B:\PICTURES*.PI3' – this would be the PATH name. However, if the name 'PICTURES' was misspelt then the path would not be found and error code –34 would be returned in register d0.

The number returned in d0 (see chapter four) is in the form $FFFFFFxx, eg $FFFFFFDF which is –33 decimal, the code for 'file not found'. Now how can we convert the negative hex numbers found in d0 when a file Open (or Save or Load) error results, to the negative decimal numbers and thus know what error actually occurred? In assembler this is quite simple, all we have to do is to use the 'neg.l d0' instruction which negates what ever is in register d0, and if the number is already negative then the result will be the positive number we want.

However the assembler will convert negative decimal numbers to hex for us:

```
* EX6.S Try to open a DEGAS file and check to see whether it
* exists

        move.w   #0,-(sp)        ; set file attribute
        move.l   #file_name,-(sp)  ; address of filename
        move.w   #$3d,-(sp)      ; open function number
        trap     #1              ; hello GEMDOS
        add.l    #8,sp
        tst      d0              ; -ve number?
        bmi      error_routine   ; yes, go to error routine
        move.w   d0,handle       ; store file handle
        bra      exit

error_routine:
* a couple of examples
        cmpi.l   #-33,d0
        beq      error_message
        cmpi.l   #-34,d0
        beq      error_message
        bra      exit

error_message:
        move.l   #error,-(sp)    ; put address of string on stack
```

```
    move.w   #9,-(sp)     ; Gemdos function 'print a line',
*'Cconws'
    trap     #1
    addq.l   #6,sp        ; correct stack

* wait for key press
    move     #2,-(sp)     ; device number (console)
    move     #2,-(sp)     ; BIOS routine number
    trap     #13     ; Call Bios
    addq.l   #4,sp
exit:
    move.w   #20,-(sp) ; leave gracefully!
    move.w   #$4c,-(sp)
    trap     #1

error:  dc.b    'Cannot find file or path not found',0
handle:         ds.w    1
file_name:      dc.b    'XX.PI3',0
```

Note to test the workings of this program 'XX.PI3' should not exist on the disk!

Note that part of EX2.S has been utilised in this example.

Most of the code should be familiar to the reader except for the following:

```
    cmpi.l   #-33,d0      ; Compare immediate data -33
    beq      error_message
```

This piece of code can be translated as compare −33 with what ever is in register d0. If −33 is the same as the contents of d0 then branch to 'error_message'. 'cmp' actually subtracts the source operand from the destination operand and sets the condition flags accordingly. However, the result of the subtraction does not affect the destination register. Thus if the comparison is equal to zero then the 'beq' instruction 'branch if equal to zero' will send the program to the 'error_message:' label.

Also:

```
        bra     exit
```

'bra' equals 'branch always' to the label in the operand field, and in this case makes sure that if the file did exist that we could exit properly as the program would branch straight to 'exit:'. If 'bra exit' was not placed there and the file had been found then the program would continue with 'error_routine', not what would be needed!

Note that it is not possible to use apostrophes in a string as the assembler would expect a null after it. So this would result in errors being reported by the assembler, for instance:

```
error:      dc.b    'Can't find file or path not found',0
```

The errors printed by the assembler were:

Pass1 (Garbage after instruction. No ';' before comment)

Assembler: line 51 (Non-terminated string)

The second error message is the more accurate one.

So far we have looked at opening a file but haven't actually opened one! However, this will be rectified soon, as on the supplied disk is a DEGAS .PI3 file (med res users should use the .PI2 file instead), called PICT1.PI3 which will be opened, loaded and closed.

(2) Reading a file and (3) closing a file:

To read a file we have to decide where we are going to place it: in a buffer until we need it, or to place it directly to the screen, which in essence is a 32K buffer.

When a file is opened a file handle is returned, however GEMDOS allocates numbers to the standard devices too which means we can use these numbers when writing to them:

- 0 standard input (usually console)
- 1 standard output (usually console)
- 2 RS-232

3 printer–standard list
6+ and up– file handles

We could also get the screen address from the Xbios function number 3 which returns the address of the screen in register d0.

In order to display a DEGAS file we need to take a closer look at how a DEGAS file is organised. When a DEGAS screen is saved to disk the first 34 bytes – known as its header – contains picture information. The first word (2 bytes) contains the resolution of the file:

1= low res
2= med res
3= hi res

The rest of the header contains the colour palette, which is made up of 16 words, with each word corresponding to a colour. The ST can display 16 colours in low res, 4 colours in med and 2 in high res (black and white). Note a 34 byte header is always used even when in high resolution despite only 2 colours being used.

The rest of the DEGAS file contains 32K or 32000 bytes of bit image information of the DEGAS screen. 'K'= Kilobytes which is a equivalent to approx 1000 bytes of data. Approximately as programmers when talking about kilobytes of RAM or data do not use it as a completely accurate measurement as 1 Kilobyte of RAM or data is actually 1024 bytes of data, or 2 to the power of 10. 'Bit image' refers to the fact that the screen whether monitor or TV is made up of pixels (PIcture ELementS), or little dots of light that are either on or off for black and white displays or are coloured for colour displays. A monitor or TV when connected to a computer displays what is held in screen RAM. As each dot of the display can be held in the screen RAM as a bit then the display is said to be bit–mapped as each pixel on the screen corresponds to a bit of screen RAM. Even text is drawn using a number of dots. An 'A' can be crudely represented like this:

```
  .
.....    or even better like this:  see diagram over the page
.   .
```

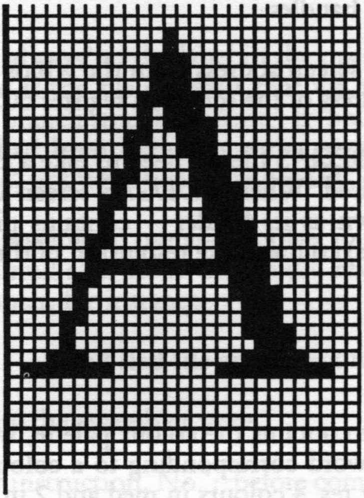

diagram 6:1 bit map of 'A'

This is why ASCII codes are used as the number is used to fetch the corresponding bit image onto the screen. Fortunately it is done so fast that when we press a key on the ST's keyboard we don't even notice a delay, although a lot of computing has to happen to get the text onto the screen. Each line of pixels across the screen is called a scan line, as the electron beam that is used in TV's and monitors scans or refreshes the screen approximately every 60 times per second by moving from the top to the bottom of the display. A high resolution DEGAS file stores each 8 pixels in each byte, so each pixel is represented by a single bit. The first byte represents the first eight pixels in the top left hand corner of the display, and each succeeding byte represent the next pixels continuing to the right. This is the way the screen is also represented in screen memory. See high and med resolution mode figures.

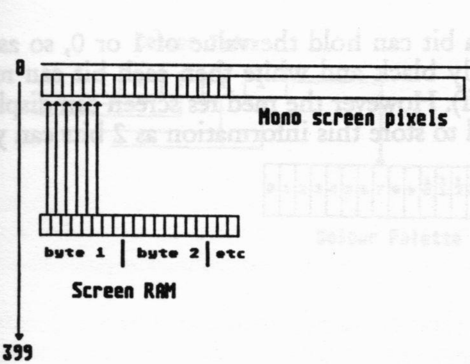

diagram 6:2 High resolution screen format

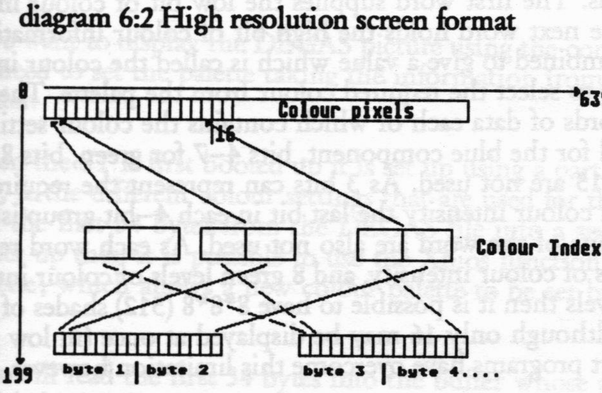

diagram 6:3 Medium resolution screen format

As a high res screen has 640 pixels or dots across then it needs 640/8 bytes to represent one (scan) line of the display, which is 80 bytes per line. The next line is represented by the next 80 bytes, and as the high res ATARI monitor has 400 dots down, then the screen can be represented by 80 bytes *400 which equals 32000 bytes, or 32K. This is why the high res monitor is said to have resolution of 640*400.

Med res= 640 * 200, 4 colours

Low res= 430 * 200, 16 colours

As has been stated before a bit can hold the value of 1 or 0, so as the high res screen displays only black and white then each bit can represent black (on) or white (off). However the med res screen can display 4 colours so 2 bits are needed to store this information as 2 bits can yield 4 possible combinations:

Bit pattern	value
00	0
01	1
10	2
11	3

Each 16 pixel group of dots on the screen is held in screen RAM as two consecutive words. The first word supplies the low bit of colour information whilst the next word holds the high bit of colour information. These bits are combined to give a value which is called the colour index which enables us to select the required colour from the palette. The palette holds 16 words of data each of which contains the colour settings. Bits 0–3 are used for the blue component, bits 4–7 for green, bits 8–11 for red. Bits 12–15 are not used. As 3 bits can represent the required 8 different levels of colour intensity the last bit in each 4–bit group is not used; the last 4 bits of the word are also not used. As each word represents 8 blue types of colour intensity, and 8 green levels of colour intensity, and 8 red levels then it is possible to have 8*8*8 (512) shades of colour on the ST, although only 16 may be displayed at once (in low res). Some software art programs have overcome this limitation however.

Similarly for low res except 16 colours can be displayed so 4 bits are needed to represent 16 colours, and 4 consecutive words in memory are needed to describe a single pixel.

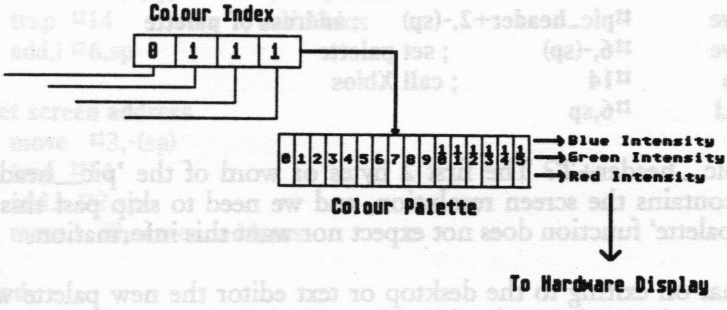

diagram 6:4 Colour palette arrangement

If we want to display the DEGAS picture using the correct colours then we need to set the palette taking the information from the DEGAS file header.

When the ST is first booted up it is set up using a particular palette setting, ie the different colour settings that are used for the desktop. If we read the first 34 bytes from the DEGAS file into a specific buffer that we set up then it is possible to use the Xbios function number six, 'setpalette', which allows a new colour palette to be set by using this routine.

This will read the first 34 bytes into the buffer whose address is held at the label 'pic_header:'

```
        move.l    #pic_header,-(sp)
        move.l    #34,-(sp)        ; number of bytes to read
        move      handle,-(sp)
        trap      #1
        add.l     #8,sp

pic_header:  ds.b 34
```

This program fragment will set the palette to the one the DEGAS file was created with:

*** use new palette**

```
        move      #pic_header+2,-(sp)     ; address of palette
        move      #6,-(sp)                ; set palette
        trap      #14                     ; call Xbios
        add.l     #6,sp
```

Why 'pic_header+2'? The first 2 bytes or word of the 'pic_header' buffer contains the screen resolution, and we need to skip past this as the 'setpalette' function does not expect nor want this information.

Note that on exiting to the desktop or text editor the new palette will continue to be used. We should really reset the palette back to what is was prior to setting the new palette. Fortunately there is a fairly easy solution this problem which will be shown in EX8.S in chapter seven.

So placing the file on the screen would be entail the following:

```
* EX7.S  Open and read a DEGAS file to the screen.

*open
        move.w    #0,-(sp)                ; set file attribute
        move.l    #file_name,-(sp)        ; address of filename
        move.w    #$3d,-(sp)              ; open function number
        trap      #1                      ; hello GEMDOS
        add.l     #8,sp
        tst       d0                      ; -ve number?
        bmi       general_error           ; yes, go to error routine
        move.w    d0,handle               ; store file handle

* read palette data

        move.l    #pic_header,-(sp)       ; pic_header address
        move.l    #34,-(sp)               ; number of bytes to read
        move      handle,-(sp)
        trap      #1
        add.l     #8,sp
        tst       d0                      ; -ve number?
        bmi       general_error           ; yes, go to error routine

* use new palette
```

```
    move.l #pic_header+2,-(sp)   ; address of palette
    move  #6,-(sp)               ; set palette
    trap  #14                    ; call Xbios
    add.l #6,sp

* get screen address
    move  #3,-(sp)
    trap  #14
    add.l  #2,sp
    move.l  d0,screen_address

* read
    move.l screen_address,-(sp) ; address of buffer
    move.l #32000,-(sp)    ; buffer size/number of
* bytes to read
    move.w  handle,-(sp)
    move.w  #$3f,-(sp)
    trap  #1
    add.l  #12,sp
    tst.l d0          ; see if error
    bmi  general_error

* close
    move handle,(sp)
    move  #$3e,-(sp)
    trap  #1
    add  #4,sp
    tst.l  d0
    bmi  general_error
    bra  wait

general_error:
* a couple of examples
    cmpi.l  #-33,d0
    beq  error_message
    cmpi.l #-34,d0
    beq  error_message
    bra  exit

error_message:
    move.l   #error,-(sp)   ; put address of string on stack
```

```
    move.w      #9,-(sp)     ; Gemdos function 'print a line',
*'Cconws'
    trap        #1
    addq.l      #6,sp        ; correct stack
wait:
* wait for key press
    move        #2,-(sp)     ; device number (console)
    move        #2,-(sp)     ; BIOS routine number
    trap        #13          ; Call Bios
    addq.l      #4,sp

exit:
    move.w      #20,-(sp)    ; leave gracefully!
    move.w      #$4c,-(sp)
    trap        #1

error:          dc.b    'An error has occurred!',0
handle:         ds.w    1
file_name:      dc.b    'PIC1.PI3',0
screen_address: ds.l 1

    .bss
pic_header:     ds.b 34
```

Chapter 7
Restoring the Palette and Files

This chapter continues where chapter six left off, and looks at restoring the palette before exiting a program. File creation (saving) is also examined.

Fortunately there is an easy way of restoring the palette after using a DEGAS file. What we need to do is to save the palette that is in use prior to altering the palette from the loaded DEGAS file. The BIOS provides a way of doing this with the call 'Setcolor', which allows the changing of a colour in a single hardware colour register. However, by passing a negative value we can read the values instead of changing them – with the result in register d0. The 'setcolor' assembly language format looks like this:

```
move      #newcolor,-(sp)
move      #register,-(sp)
move      #7,-(sp)
trap      #14
addq.l    #6,sp
```

Where register is a number from 0 to 15, and newcolor is a word containing 0–$777. See previous chapter.

So all we have to do is to set up a loop to read each palette setting and before we exit reset the palette with 'Setpalette', Bios call 6.

```
* this program fragment reads the colour palette from the colour
* registers into a buffer called 'palette_buffer'.

    move.l #palette_buffer,a3
    move.l #15,d3

read_again:

    move  #-1,-(sp)     ; read contents
    move  d3,-(sp)      ; counter from 15 to 0
```

```
        move        #7,-(sp)      ; 'Setcolor'
        trap        #14
        addq.l      #6,sp

        move        d0,(a3)+      ; place contents of d0 in palette_buffer
        sub         #1,d3
        cmpi.b      #0,d3
        bge         read_again

palette_buffer:  ds.w 16
```

This program fragment sets up a loop so that the 'setcolor' call can be accessed 15 times. Each time the trap is called d3 is reduced by 1 by the instruction 'sub #1,d3'. Then the contents of d3 are compared to zero, 'cmpi.b #0,d3'. The mnemonic 'cmpi.b' means 'compare immediate data'. If d3 is not zero then the 'bge' mnemonic sends it back to the 'read__again:' label. 'bge' means 'branch if greater than or equal to zero'.

After the trap has been called register d0 contains the information we need so it is placed in the place pointed to by the address in register a3. In other words a3 contains 'palette__buffer' address and the operands 'd0,(a3)+' says find the address in a3 and put d0 there, and then increment that address by a word. This could have also been written like this:

```
        move        d0,(a3)
        add         #2,a3
```

but (a3)+ does the same job. Note that adding 1 to a3 would increment it by a byte, adding 2 increments a3 by a word (2 bytes), and adding 4 increments a3 by a long (4 bytes or 2 words).

```
* EX8.S  Open and read a file to a buffer, display DEGAS file
* to screen. Reset the palette before exiting.

* get palette
        move.l      #palette_buffer,a3
        move.l      #0,d3

read_again:
```

```
        move      #-1,-(sp)       ; read contents
        move      #3,-(sp)        ; counter from 15 to 0
        move      #7,-(sp)        ; Setcolor
        trap      #14
        addq.l    #6,sp

        move      d0,(a3)+        ;  place  contents  of  d0  in
* palette_buffer
        add       #1,d3           ; counter
        cmpi.b    #16,d3
        bne       read_again

*open
        move.w    #0,-(sp)        ; set file attribute
        move.l    #file_name,-(sp)  ; address of filename
        move.w    #$3d,-(sp)      ; open function number
        trap      #1              ; hello GEMDOS
        add.l     #8,sp
        tst       d0              ; -ve number?
        bmi       general_error      ; yes, go to error routine
        move.w    d0,handle       ; store file handle

* read palette data

        move.l    #pic_header,-(sp)  ; pic_header address
        move.l    #34,-(sp)       ; number of bytes to read
        move      handle,-(sp)
        move.w    #$3f,-(sp)
        trap      #1
        add.l     #12,sp
        tst       d0              ; -ve number?
        bmi       general_error      ; yes, go to error routine

* get current screen res

        move      #4,-(sp)
        trap      #14
        addq.l    #2,sp
* res returned in d0
```

```
    move        pic_header,d1   ; get res of DEGAS file
    cmp         d1,d0   ; compare to actual res in
* use
    bne         error_message

* use new palette
    move.l      #pic_header+2,-(sp)    ; address of palette
    move        #6,-(sp)          ; set palette
    trap        #14               ; call Xbios
    add.l       #6,sp

* get screen address
    move        #3,-(sp)
    trap        #14
    add.l       #2,sp
    move.l      d0,screen_address

* read
    move.l      screen_address,-(sp) ; address of buffer
    move.l      #32000,-(sp)    ;  buffer  size/number of
* bytes to read
    move.w      handle,-(sp)
    move.w      #$3f,-(sp)
    trap        #1
    add.l       #12,sp
    tst.l       d0          ; see if error
    bmi         general_error

* close
    move        handle,(sp)
    move        #$3e,-(sp)
    trap        #1
    add         #4,sp
    tst.l       d0
    bmi         general_error
    bra         wait

general_error:
* a couple of examples
```

```
    cmpi.l   #-33,d0
    beq      error_message
    cmpi.l   #-34,d0
    beq      error_message
    bra      exit

error_message:
    move.l   #error,-(sp)  ; put address of string on stack
    move.w   #9,-(sp)      ; Gemdos function 'print a line',
*'Cconws'
    trap     #1
    addq.l   #6,sp         ; correct stack
wait:
* wait for key press
    move     #2,-(sp)      ; device number (console)
    move     #2,-(sp)      ; BIOS routine number
    trap     #13           ; Call Bios
    addq.l   #4,sp
exit:

* reset palette
    move.l   #palette_buffer,-(sp) ; address of old palette
    move     #6,-(sp)              ; set palette
    trap     #14                   ; call Xbios
    add.l    #6,sp

    move.w   #20,-(sp)  ; leave gracefully!
    move.w   #$4c,-(sp)
    trap     #1

error: dc.b    'An error has occurred!',0
handle:        ds.w   1
file_name:     dc.b   'PIC1.PI3',0
screen_address: ds.l 1
palette_buffer: ds.w 16
    .bss
pic_header:    ds.b 34
```

What if we wanted to load the DEGAS file from another disk drive or from another path or both. For instance say PIC1.PI3 was on drive B, then we should write:

```
file_name:      dc.b 'B:\PIC1.PI3',0
```

Then the program would go to the correct drive and load PIC1.PI3 from there. Drive A: would not be accessed. But what if the DEGAS was in a folder called 'PICS'. Then we would write:

```
file_name:      dc.b 'B:\PICS\PIC1.PI3',0
```

This time the program would try to load the DEGAS file from the folder 'PICS', on drive B:. Of course if the folder or the file did not exist then an error would be returned in register d0.

You may be wondering why the GEM file selector box has not been used to select a DEGAS picture. The answer to this question is that to use anything connected with GEM we first need to do quite a bit of setting–up which will be looked at in the next chapter.

This part of the program neatly illustrates the difference between using a label as an address and the contents pointed to by the address of that label:

```
    move        pic_header,d1       ; get res of DEGAS file
    cmp         d1,d0               ; compare to actual res in use
    bne         error_message
```

'move pic__header,d1' puts the contents of the address referenced by the label 'pic__header' into d1. Prior to this the current resolution being used was placed in d0. They are compared to each other – if they are not equal ('bne' means branch if not equal) then the program will branch to 'error__message'. Loading the incorrect resolution DEGAS screen will not help at all! And this test helps to bypass this possibility.

If 'move.l #pic__header,d1' is used then the actual address of the label 'pic__header' would be placed in d1. However, by using 'move pic__header,d1' the (word) contents pointed to by the address 'pic__header' are placed in d1. This is extremely useful as most assembly language programming makes use of this feature, and it can lead to some elegant programming solutions.

If 'move pic_header+2,d1' is used then the contents of the address 'pic_header' plus 2 bytes would be placed in d1. Note that the address accessed by this method must always be on an even boundary so that 'move.l #pic_header+3,d1' would result in an address error (3 bomb icons on screen) as odd addresses cannot be accessed. Similarly, 'move pic_header+3,d1 would also result in 3 bombs on the screen. However, 'move.b pic_header+3,d1' is ok as in this case only a byte is fetched and although the address is odd we are accessing the contents not trying to refer to an address per se.

Whenever bombs appear on the screen the program has invariably terminally 'crashed', and the result is either the ST will 'hang up', ie the mouse pointer, keyboard and screen will freeze or if you are lucky you will be returned to the desktop or the calling program. Even if you are returned to the calling program it may still be necessary to cold–boot your ST either by the off/on switch, or soft–boot it by pressing the reset button on the back at the left of the ST. See chapter 21 for a list of 'exception handling' whenever severe program errors occur.

Writing a file to disk:

This process is very similar to loading a file from disk.

Note that in the following example the current palette is first placed in a palette buffer, and then the screen address is found, and the current resolution of the screen is also found. These factors are needed for the DEGAS header. As 32000 bytes are saved to disk a check as to whether this actually happens is done. This is quite easy as when a file is written to disk the amount that is actually saved to disk is returned in d0, after the writing to disk has finished. It is easy to check the amount returned with the amount that was wanted to be saved and report an error to the user advising a full disk. In the example the 'general_error' routine is used if a full disk is found but a more specific error message would, of course, be used in practise. This time a friendly message advising the user to press any key is placed on screen, followed by the GEMDOS call 'Crawcin', which waits for any key to be pressed but does not echo (show) it to the screen.

```
* EX9.S Save and Close a 32K file (screen) to disk in DEGAS
* format. Also check to see if disk is full after 32K is saved.

start:
      move.l      #pic_head+2,a3    ; place buffer address in a3
      move.l      #0,d3             ; counter

* get contents of palette

read_again:

      move        #-1,-(sp)     ; read contents
      move        d3,-(sp)      ; counter from 15 to 0
      move        #7,-(sp)      ; Setcolor
      trap        #14
      addq.l      #6,sp

      move        d0,(a3)+      ;  place contents of d0 in
* palette_buffer
      add         #1,d3
      cmpi.b      #17,d3
      bne         read_again

*  get screen address
      move        #3,-(sp)
      trap        #14
      add.l       #2,sp
      move.l      d0,screen_address

* create file
      move        #0,-(sp)          ; read/write status
      move.l      #file_name,-(sp)  ; address of filename
      move        #$3C,-(sp)
      trap        #1
      addq.l      #8,sp
      tst         d0
      bmi         general_error
      move        d0,handle         ; get handle in 'handle'

* get current screen res
      move        #4,-(sp)
```

```
    trap        #14
    addq.l      #2,sp
* res returned in d0

* write  palette file
    move        d0,pic_head
    move.l      #pic_head,-(sp)    ; address of buffer
    move.l      #34,-(sp)          ; number of bytes to save
    move.w      handle,-(sp)
    move.w      #$40,-(sp)
    trap        #1
    add.l       #12,sp
    tst.l       d0
    bmi         general_error

* write 32K screen RAM file
    move.l      screen_address,-(sp)   ; address of buffer
    move.l      #32000,-(sp)           ; 32K of RAM to save
    move.w      handle,-(sp)
    move.w      #$40,-(sp)
    trap        #1
    add.l       #12,sp
    tst.l       d0
    bmi         general_error
    move.l      d0,saved_amount

* close file
    move        handle,(sp)
    move        #$3e,-(sp)
    trap        #1
    add         #4,sp
    tst.l       d0
    bmi         general_error

    cmpi.l      #32000,saved_amount ; see if 32000 bytes were
* saved
    blt         general_error      ; if not then disk full
    bra         wait

general_error:
```

```
* a couple of examples
    cmpi.l      #-33,d0
    beq         error_message
    cmpi.l      #-34,d0
    beq         error_message
    bra         exit

error_message:
    move.l      #error,-(sp)   ; put address of string on stack
    move.w      #9,-(sp)       ; Gemdos function 'print a line',
*'Cconws'
    trap        #1
    addq.l      #6,sp          ; correct stack
wait:
    move.l      #message,-(sp) ; put address of string on stack
    move.w      #9,-(sp)       ; Gemdos function 'print a line',
*'Cconws'
    trap        #1
    addq.l      #6,sp          ; correct stack

* wait for key press
    move        #7,-(sp)       ; Get key, no echo
    trap        #1             ; Call GEMDOS
    addq.l      #2,sp

exit:
    move.w      #20,-(sp)  ; leave gracefully!
    move.w      #$4c,-(sp)
    trap        #1

error:          dc.b    'An error has occurred!',0
message:        dc.b    'Press any key to exit',0
handle:         ds.w    1
file_name:      dc.b    'SAVE_PIC.PI3',0
screen_address: ds.l 1
palette_buffer: ds.w 17
    .bss
pic_head:       ds.b 34
saved_amount:   ds.l 1
```

There are a couple of interesting features in this source code, checking for a full disk is one:

```
    cmpi.l    #32000,saved_amount ; see if 32000 bytes were saved
    blt       general_error       ; if not then disk full
```

Here the amount that we expect to be saved 32000 bytes, is compared to the actual amount saved. Previously the actual amount saved was placed in 'saved_amount', and after the file is closed the check is made.

Also by finding the actual screen res in use at the present time we can then use this result to place in the DEGAS header:

```
* get current screen res

    move      #4,-(sp)
    trap      #14
    addq.l    #2,sp
* res returned in d0

* write palette file
    move      d0,pic_head
```

'move d0,pic_head' will place the current resolution at the start of the header which is where DEGAS expects to find it. Without this value here DEGAS will not load the file.

Strictly speaking all the example programs so far have been TOS programs, usually identifiable from the '.TOS' extension (although '.PRG' has been used for convenience). TOS programs do not make use of GEM or the mouse, which is what we have been doing, although it does not really seem to matter what extension is given with such small examples.

There are a couple of interesting features in this source code, checking for a full disk is one:

```
cmpi.l   #32000,saved_amount   ; see if 32000 bytes were saved
blt      general_error         ; if not then disk full
```

Here the amount that we expect to be saved 32000 bytes, is compared to the actual amount saved. Previously the actual amount saved was placed in 'saved_amount', and after the file is closed the check is made.

Also by finding the actual screen res in use at the present time we can then use this result to place in the DEGAS header:

```
* get current screen res
     move     #4,-(sp)
     trap     #14
     addq.l   #2,sp
* res returned in d0

* write palette file
     move     d0,pic_head
```

'move d0,pic_head' will place the current resolution at the start of the header which is where DEGAS expects to find it. Without this value here DEGAS will not load the file.

Strictly speaking all the example programs so far have been TOS programs, usually identifiable from the '.TOS' extension (although '.PRG' has been used for convenience). TOS programs do not make use of GEM or the mouse, which is what we have been doing, although it does not really seem to matter what extension is given with such small examples.

Chapter 8
Mono Pics to Low Res

This chapter looks in detail at converting a high resolution DEGAS file to low res and displaying it on screen.

btst and bset

As the 'btst' – bit test and 'bset' – bit set instructions are used extensively in the source code it would be useful to look at these before proceeding.

The 'btst' instruction allows the programer to test any bit in a register whether it is the actual contents of the register or a pointer to an area of memory, eg 'btst #15,(a0)'. Here the bit tested is bit 15 (counting from 0), and in an area of memory referenced by the address register a0. The parentheses around a0 tell us that we want to refer to the contents of the address held in register a0.

If register d0 held the hex value $0f – (%00000000000000000000000000001111), but we did not know this, then a simple 'btst #0,d0' would test bit 1 for us. By using the branch if equal (to zero) 'beq' instruction immediately after the test then we could tell what the result of the test was. The zero flag in the ccr would be set to one if the bit tested was zero. In this case the result would be that the ccr zero flag would be not be set to 1 and and the branch if equal to zero condition would be not be fulfilled.

'btst' using immediate data on a register can only be used as 'btst.l', whilst testing a bit in memory only a 'btst.b' is allowed. The assembler defaults to these values when they are not specified.

To set a bit (to one) in a register or in memory, 'bset' is used in a similar way that 'btst' is described above. It also has the same byte and long restrictions.

Converting

Sometimes you may wish to convert mono pictures into colour and vice–versa. It is in operations like this that assembly language comes into it's own – high level languages such as Basic and C cannot compete with the sheer speed that is offered to the programmer by writing in assembly language.

This routine loads a high–res DEGAS picture from a disc and converts it into low resolution, displaying it on the screen. Obviously, this routine must be run in low–res – otherwise garbage will result!

If you were to look closely at part of a low–res screen and compare it to a high–res screen, you would see that low–res pixels are four times the size of high–res pixels

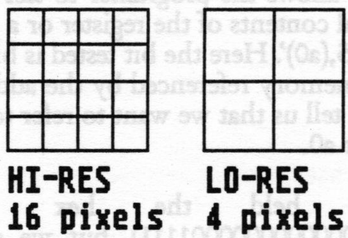

diagram 8:1

The basis of this high to low–res conversion is that we take a grid of four high–res pixels and convert them into one low–res pixel.

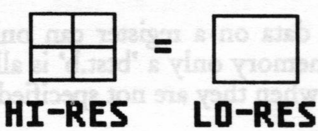

diagram 8:2

On a high–res picture any pixel can have one of two states – on or off

– black or white. Some areas of high–res pictures may appear to be different shades of grey, this is because the eye cannot perceive the individual pixels but instead perceives the density of black pixels making us think that we see an area of grey.

With a low–res picture things now begin to get rather more complicated. Instead of just being on or off a pixel can now have one of sixteen different values or colours and each colour is made up of three indices – one for red, one for green and one for blue! The actual values for each colour are stored in an area of RAM called the palette. If the ST finds that a pixel is set to be, for example, colour number 10 it looks into the palette and finds the exact amount of red, green and blue light to tell the monitor to transmit.

Setting the palette

There are two XBIOS calls that allow the programmer to set the palette colours – XBIOS 6 and XBIOS 7. Both of these calls require that one of the parameters passed is the colour (RGB values) that you require. This parameter must be passed as a word. The easiest way of doing this is to use a 3 digit hexadecimal figure – the first figure corresponds to red, the second to green and the third to blue. On a ST each figure may have a value between 0 and 7 inclusive – the higher the number, the brighter that colour. However, the STE can display many more colours.

Examples $700 = red
$007 = blue
$070 = green
$077 = yellow

In practice, using this system, if equal amounts of red, green and blue light are mixed, the resulting colour will be black, white, or a shade of grey.

This routine firstly sets the first five colours of the palette as per this table:–

Palette number/ source pixels black	Palette RGB	Resulting colour
0	$000	White

1	$222	Light grey
2	$444	Mid grey
3	$666	Dark grey
4	$777	Black

Next a 2 by 2 grid of the high–res picture is sampled and the density of black pixels is calculated by simply counting them. The value returned from this calculation is used as the colour number for the corresponding pixel that is to be set in the low–res picture.

EG. If the pixel count finds that 3 out of the 4 pixels are black (which would appear to be a dark grey) the resulting low–res pixel is set to be colour number 3 which is also dark grey.

In this way fout high res pixels are converted into a single low–res pixel.

Bits and planes

The whole operation is made slightly more difficult due to the fact that the ST holds different resolution screens in memory in different ways. The high–res screen is a simple bitmap – each bit corresponds to one pixel.

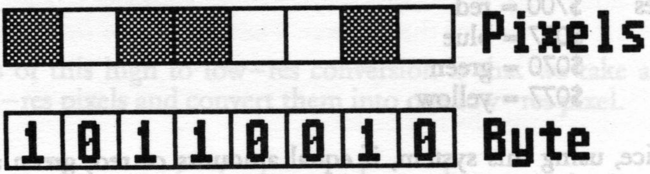

diagram 8:3

The low–res screen is much more difficult. As there are a total of 16 possible colours (0 to 15) that each pixel could be, four bits are required to determine the colour of each pixel. This is due to the fact that if the number 15 is represented as a binary number (15=%1111), four binary digits and therefore four bits are required (remember: the term 'bits' means binary digits). This would be fine if these four bits were held consecutively in memory, but they're not!

The ST's video display uses a series of 'planes'. A high–res screen needs 1 plane, a med–res needs 2 and a low–res 4. The amount of planes equals the amount of bits required to represent the largest possible colour number.

high–res	2 colours (%0–%1)	1 plane required
med–res	4 colours (%0–%11)	2 planes required
low–res	16 colours (%0–%1111)	4 planes required

A word of data for the first plane is followed by a word of data for the next plane and so on. In a low–res screen:–

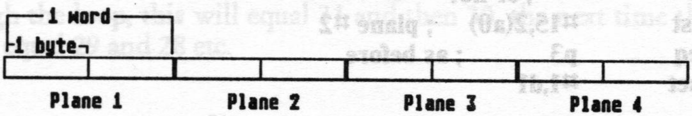

diagram 8:4

To find the colour of any one pixel we must test the relevant bits from each plane. For example to find the colour of the first pixel on the screen, we must take the first four words and test bit 31 of each of these words.

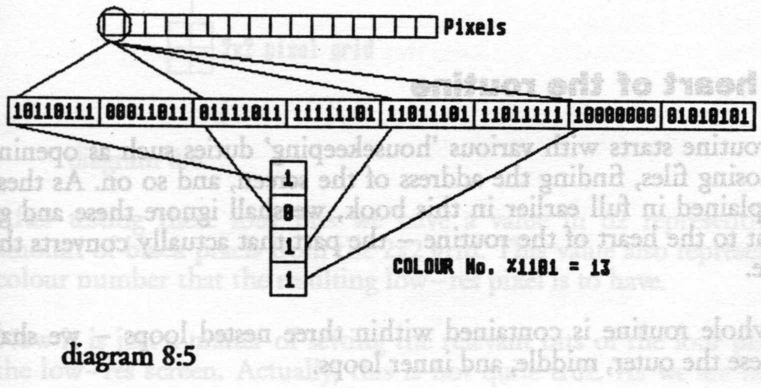

diagram 8:5

```
* EXAMPLE1.S
; Finding the colour of the top left pixel of a low-res screen

     ; get screen address

     move.w    #3,-(sp)     ; opcode
     trap      #14          ; XBIOS
     addq.l    #2,sp        ; tidy stack
     move.l    d0,a0        ; the screen address

     clr.l     d1           ; make some space
     btst      #15,(a0)     ; plane #1
     beq       p2           ; is it set?
     bset      #0,d1        ; yes
p2:                         ; or no?
     btst      #15,2(a0)    ; plane #2
     beq       p3           ; as before
     bset      #1,d1
p3:
     btst      #15,4(a0)    ; plane #3
     beq       p4
     bset      #2,d1

p4:
     btst      #15,6(a0)    ; plane #4
     beq       end
     bset      #0,d1
end:                        ; the colour number of the first
                            ; is now in register d1
```

The heart of the routine

This routine starts with various 'housekeeping' duties such as opening and closing files, finding the address of the screen, and so on. As these are explained in full earlier in this book, we shall ignore these and go straight to the heart of the routine – the part that actually converts the picture.

The whole routine is contained within three nested loops – we shall call these the outer, middle, and inner loops.

The OUTER LOOP processes a horizontal line (known as a scan line).

The MIDDLE LOOP reads two long words (64 pixels) from the high-res picture and writes four words (16 pixels) to the low-res picture.

The INNER LOOP is the actual conversion process.

It is worth taking a closer look at the MIDDLE and INNER loops. If we start at the beginning of the picture, we read the first long into d0 and another long, 80 bytes after the start of the picture, into d1. This offset of 80 bytes allows us to read the second line of the picture – this is required because we sample a 2x2 grid of pixels.

A value of 31 is moved into d4 – the comment calls this a 'bit counter'. This the number of the bit to test of registers d0 and d1. The first time through the loop, this will equal 31 and then 30, the next time through is will equal 29 and 28 etc.

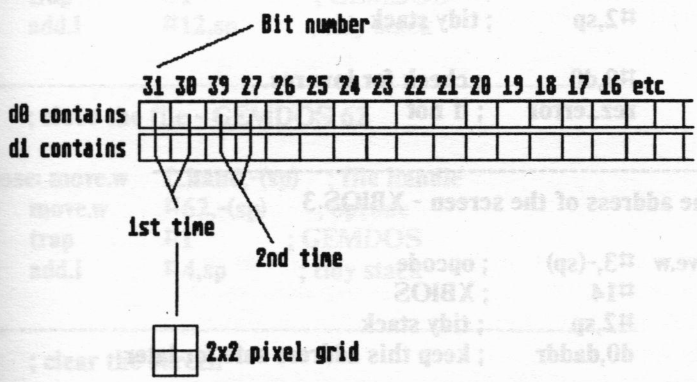

diagram 8:6

After testing these four bits we have a value in d2 representing the amount of black pixels from the 2x2 grid. This value also represents the colour number that the resulting low-res pixel is to have.

Now it is just a matter of setting the relevant bits of the four planes of the low-res screen. Actually, this is not quite true. As we are only using colours 0–4 we will never have to use the fourth plane. To represent the number 4, we only need to use 3 binary digits. So, in practice, we just ignore the 4th plane leaving it empty (equal to 0).

As we are converting two lines of 32 pixels from the high–res screen into one line of 16 pixels of the low–res screen we need to divide our 'bit counter' by two. This gives us the number of the bit to set in each of the 4 planes of the low–res picture.

```
* EX10.S
* This program converts a high res DEGAS file to a low res
* DEGAS file and displays it on the screen.


        ; check screen resolution - XBIOS 4

rez:    move.w      #4,-(sp)        ; opcode
        trap        #14             ; XBIOS
        add.l       #2,sp           ; tidy stack

        cmpi        #0,d0           ; check for low-res
        bne         rez_error       ; if not
* ----------------------------------------------------------------

        ; get the address of the screen - XBIOS 3

screen: move.w      #3,-(sp)        ; opcode
        trap        #14             ; XBIOS
        addq.l      #2,sp           ; tidy stack
        move.l      d0,daddr        ; keep this address safe for later

* ----------------------------------------------------------------

        ; open a file - GEMDOS 61

open:   move.w      #0,-(sp)        ; mode - read only
        move.l      #fname,-(sp)    ; address of a string containing
                                    ; path and filename
        move.w      #61,-(sp)       ; opcode
        trap        #1              ; GEMDOS
        addq.l      #8,sp           ; tidy stack
```

```
        move.w     d0,f_hand      ; a handle is returned in d0

        ; test for errors

        tst        d0             ; check for negative number
        bmi        file_error

* -----------------------------------------------------------------

        ; read a block from the opened file - GEMDOS 63

read:   move.l     #saddr,-(sp)   ; address of buffer to hold the
                                  ; information
        move.l     #32034,-(sp)   ; amount of bytes to read
                   ; a DEGAS file consists of a
                   ; 34 byte header followed by 32000
                   ; bytes of bitmapped data
        move.w     f_hand,-(sp)   ; file handle
        move.w     #63,-(sp)      ; opcode
        trap       #1             ; GEMDOS
        add.l      #12,sp         ; tidy stack

* -----------------------------------------------------------------

        ; close the file - GEMDOS 62

close:  move.w     f_hand,-(sp)   ; file handle
        move.w     #62,-(sp)      ; opcode
        trap       #1             ; GEMDOS
        add.l      #4,sp          ; tidy stack

* -----------------------------------------------------------------

        ; clear the screen

cls:    clr.l      d0             ; make it equal 0
        move.l     daddr,a1       ; address of the screen
        move.w     #7999,d1       ; 8000 words = 32000 bytes
cls2:   move.l     d0,(a1)+       ; move to the screen address and
                   ; increment address for the next
                   ; time around
        dbra       d1,cls2        ; round and round

* -----------------------------------------------------------------
```

```
        ; Now set the first 5 palette colours to our chosen
        ; shades of grey.  Rather than setting each colour
        ; individually using XBIOS 7 we shall set the all at
        ; once using XBIOS 6

s_pal:  move.l      #pal,a0         ; address of our buffer
        move.w      #$777,(a0)+     ; white
        move.w      #$666,(a0)+     ; light grey
        move.w      #$444,(a0)+     ; mid grey
        move.w      #$222,(a0)+     ; dark grey
        move.w      #$000,(a0)+     ; black

        move.l      #pal,-(sp)      ; buffer address
        move.w      #6,-(sp)        ; opcode
        trap        #14             ; XBIOS
        add.l       #6,sp           ; tidy stack

* ----------------------------------------------------------
        ; the conversion routine itself

        movea.l     #saddr,a0       ; the address of the picture data
        adda.l      #34,a0          ; skip the 34 byte header
        movea.l     daddr,a1        ; destination address - the screen

* THE OUTER LOOP

            ; Two scan-lines are processed at one time in
            ; the outer loop.  A high-res screen consists of
            ; 400 horizonta lines - this leads to the figure
            ; (400/2)-1=199, the -1 is needed for the decrement
            ; and branch if false (dbra) at the end of this loop

        move        #199,d2
l2:

* THE MIDDLE LOOP

            ; In the middle loop a long is processed in one pass.
            ; A high-res scanline consists of 80 bytes which equals
            ; 20 words.  As before subtract 1 to get the
```

```
        ; figure - 19.

        move        #19,d3
l:
        move.l      (a0),d0         ; get a long for processing
        move.l      80(a0),d1       ; and one from the next scanline
        adda.l      #4,a0           ; make it point to the next long for
                                    ; the next pass

        clr.l       d5              ; avoid errors!
        clr.l       d6
        clr.l       d7

        move.w      d2,-(sp)        ; we need these registers, so keep
        move.w      d3,-(sp)        ; their contents safe for later

* THE INNER LOOP

            ; now we count the number of black pixels in each
            ; 2x2 grid. Register d2 is our counter, d4 contains
            ; the number of the bit to test.

            ;   ---------
            ;   |1|3|   pixel grid (high-res)
            ;   |-----|
            ;   |2|4|
            ;   ---------
            ;

        move        #31,d4          ; bit counter
bl:
        clr.l       d2              ; clear it
        btst        d4,d0           ; test grid #1
        beq         x1              ; if white do nothing
        addq        #1,d2           ; if black increase the count
x1:     btst        #4,d1           ; test grid #2
        beq         x2              ; if white
        addq        #1,d2           ; if black
x2:     subq        #1,d4           ; decrement the bit counter
```

```
     btst      d4,d0      ; test grid #3
     beq       x3         ; as before
     addq      #1,d2
x3:  btst      d4,d1      ; test grid #4
     beq       x4
     addq      #1,d2
x4:

          ; The amount of black pixels is now held in
          ; register d2.

          ; As we are turning a long from the high-res picture
          ; into a low-res word (x4 for each plane) we need
          ; to know which bit to set.

     move.l    d4,d3      ; put the bit counter into d3
     divs      #2,d3      ; and divide by 2 to find the
                          ; low-res bit to set

          ; The amount of black pixels in the grid is held
          ; in d2. This number also happens to be the palette
          ; number of the required colour. Normally we would
          ; have 4 planes to worry about, but as we only use
          ; colours 0 to 4 (in binary %0 to %100) we only need
          ; 3 planes. The registers d5-7 are used for the
          ; output.

     btst      #2,d2      ; plane 3
     beq       c1         ; if it is not set - do nothing
     bset      d3,d7      ; else set the bit
c1:  btst      #1,d2      ; plane 2
     beq       p1         ; as before
     bset      d3,d6
p1:  btst      #0,d2      ; plane 1
     beq       p2
     bset      d3,d5
p2:
     subq      #1,d4      ; decrement the high-res bit counter

     cmpi      #0,d4      ; have we finished a high-res long?
     bge       bl         ; no
```

```
* END OF INNER LOOP                  ; yes!

    move     d5,(a1)+     ; move plane 1 to the screen
    move     d6,(a1)+     ;  "    "  2  "  "   "
    move     d7,(a1)+     ;  "    "  3  "  "   "
    adda     #2,a1        ; and compensate for the unused plane 4

    move.w   (sp)+,d3     ; recover our loop counters
    move.w   (sp)+,d2

    dbra     d3,l         ; have we finished a scanline
                          ; yes
* END OF MIDDLE LOOP

    adda     #80,a0   ; as we take 2 lines at a time from the
                      ; high-res picture, be sure to miss a line

    dbra     d2,l2    ; have we finished the lot?
                      ; certainly have
* END OF OUTER LOOP

* ------------------------------------------------------------
    ; now wait for a key press BIOS 2

kp: move.w   #2,-(sp)     ; device code - the console
    move.w   #2,-(sp)     ; opcode
    trap     #13          ; BIOS
    add.l    #4,sp        ; tidy stack
* ------------------------------------------------------------
    ; and quit cleanly GEMDOS 76

quit: clr.w  -(a7)        ; status code
    move.w   #76,-(a7)    ; opcode
    trap     #1           ; GEMDOS
* ------------------------------------------------------------

    ; error trapping routines
    ; using GEMDOS 9 - print a line of text to the screen
```

```
rez_error:                          ; wrong resolution
      move.l     #error1,-(sp)  ; address of the message
      bra        err_cont       ; go and finish the call

file_error:                         ; error opening file
      move.l     #error2,-(sp)  ; address of the message

err_cont:
      move.w     #9,-(sp)       ; opcode
      trap       #1             ; GEMDOS
      add.l      #6,sp          ; tidy stack
      bra        kp             ; forget it!

* -----------------------------------------------------

fname:           dc.b "monapipe.pi3",0    ; file name and path

      ; error messages

error1: dc.b   " ERROR - low res only!",0

error2:          dc.b  " DISC READ ERROR - CANNOT"
                 dc.b  "CONTINUE!",0

* -----------------------------------------------------

         .bss
f_hand:          ds.w   1          ; file handle
pal:             ds.w   16         ; buffer for our palette
saddr:           ds.b   32034      ; buffer for the picture
daddr:           ds.l   1          ; screen address
```

Chapter 9
Formatting a Disk

This chapter looks at, in some detail, on how to format a floppy disk.

Formatting a disk can be a very complicated procedure – some programmers make their living from devising disk formats, especially in the computer games field where the copy–protection is quite often based on the disk format. However, formatting a 'normal' disk is relatively simple due to the fact that Atari's operating system provides us with some calls (traps) that will do most of the work for us. These calls are:

XBIOS $0A	_flopfmt	format track
XBIOS $12	_protobt	prototype boot sector image
XBIOS $09	_flopwr	write sector(s) to floppy disk
BIOS $04	_rwabs	read/write sectors to device

To understand how these calls work, it is necessary to learn something about the nature of a floppy disk.

When we format a disk, all we are doing is writing a code to the disk so that when the computer later writes data to the disk it can find it again later. The disk is divided into a series of concentric rings called 'tracks' and then each track is further divided into 'sectors'.

Normally a sector can contain 512 bytes of data. From this it is easy to calculate how many bytes in total a disk can hold. If we take a normal single–sided disk which is formatted with 80 tracks and each track having 9 sectors –

80 x 9 = 720 sectors in total

720 x 512 = 368640 bytes

However, not all this space can be used for storing data as the computer reserves the first two sectors for it's own use. This is where the 'boot sector', 'directories' and 'file allocation tables' (FAT) are to be found. A disk stores data in a similar way to a book. A book will typically have numbered pages, a contents section and an index.

Numbered pages	=	Sectors
Contents section	=	Directories
Index	=	File allocation table (FAT)

Directories – store information about files such as the file name, attributes (whether read only, hidden etc.) time and date of creation, length in bytes etc.

File allocation tables – tell the computer where to find the data for each file, ie what sectors on the disk that it occupies.

So far we have not looked at the boot sector, this contains something known as the 'bios parameter block'. The bios parameter block contains information about the format of a disk such as the amount of sectors per track, total amount of sectors on disk, amount of sides etc. It also contains the disk's serial number. The computer uses the disk's serial number to determine if a disk has been changed, therefore each disk should have an unique serial number. This is normally achieved by using a random number. The boot sector can also contain executable code (a program), this can be anything from a legitimate program such as a loader program (which makes a disk self–booting) to a virus!

It is beyond the scope of this book to go any deeper into the mysteries of disk formats, FATs, boot sectors and boot sector code. However there are books on the market that deal with these subjects specifically and in detail. This aspect of computer programming is fascinating and you will probably find further study very rewarding.

To format a floppy disk these steps must be taken:

1. Format each track in turn
2. Prototype a bootsector and write it to the disk
3. Make the FATs.

Let's look at each step in detail.

1. Format each track in turn

This uses the XBIOS call – *_flopfmt*. The following parameters must be passed

a. *fcod* – format code (sometimes know as virgin). Determines what value the sectors will hold after formatting. Normally $E5E5.

b. *magc* – magic number. A contant used during formatting. This must be set to $87654321.

c. *intl* – interleave. Determines the order of sectors within each track. Normally set to 1.

d. *sidn* – side number. Side number to format, either 0 or 1.

e. *trkn* – track number. Track number to format.

f. *sptk* – sectors per track. 9 is normal

g. *devn* – device number. Either 0 for drive A or 1 for drive B.

h. *scrt* – not used so set to 0

i. *buff* – buffer address. This call uses a buffer in which to prototype the format before writing it to the disk. Available documentation says the 8k must be reserved for a normal disk with 9 sectors per track. Our routine reserves 10k just to be on the safe side.

As this 'trap' is called many times, the obvious thing to do is to use a loop. A BASIC programmer would write something like this:

```
FOR track_number = 0 TO 79
(format routine)
NEXT track_number
```

As the track number is increasing we will not be able to use a decrement and branch if false (dbra), so we must devise our own looping system:

```
    move.w    #0,d7    ; track_number
loop:
    (format routine)

    addq.w    #1,d7    ;increase track_number
    cmpi.w    #80,d7   ;have we finished?
    blt       loop     ;if no
; blt= branch if less
```

A further bit of trickery involving the format code (virgin) is required when formatting each track. As mentioned earlier the computer reserves the first two tracks of a disk for directories, FAT's and the bootsector. These tracks should be zeroed whereas all the rest of the tracks should contain the standard filler – $e5e5. To do this we use register d6 to hold the format code. At the start of the routine this register is cleared (set to zero). After two tracks (0 and 1) have been formatted, the value $e5e5 is moved into it.

```
    clr.l     d6       ; format code (virgin)
    move.w    #0,d7    ; track number to format

    (format routine - _flopfmt call)

    addq.w    #1,d7    ; increase track number
    cmpi.w    #2,d7    ; two tracks formatted
    bne       no_change ; if not
    move.w    #$e5e5,d6 ; else change the format code
no_change:
```

2. Prototype a boot sector and write it to disk.

To prototype the bootsector we use the XBIOS call – _protobt, which requires the following parameters:

a. *exfl* – executable flag. 1=executable 0=non–executable. Normally set to 0.

b. *dskt* – disk type.
    0=40 trk S/S
    1=40 trk D/S
    2=80 trk S/S

3=80 trk D/S

Either set to 2 or 3 (1 and 2 are used for IBM format disks). You will see from this that if you wish to format a disk with a non-standard amount of tracks (ie. 81) you will not be able to use this call and must make your own bootsector.

c. *sern* – serial number. According to the Atari documentation if a number greater than 24 bits is used (>$1000000) a random number is generated. In the routine at the end of this chapter a random number is generated using the XBIOS $11 call – *_random*

```
move.w    #$11,-(sp)  ; opcode
trap      #14         ; XBIOS
addq.l    #2,sp       ; tidy stack
```

This call returns a random number in register d0.

d. *buff* – pointer to a 512k buffer. When this trap has been called the prototyped bootsector will be found at the address of this buffer. It is then a simple matter of writing it to the disk.

To write the bootsector to the disk we must know which sector to write it to. Simple! The bootsector is always the first sector on the disk – Side 0, Track 0, Sector 1. To write the bootsector to the disk the XBIOS call _flopwr is used. You may notice that the formatting program at the end of this chapter also uses a BIOS call (rwabs) to write sectors to the disk, so why do we not use this call to write the bootsector as well? The answer to this is because Atari tell you not to! No explanation is given but using rwabs to write the bootsector to a disk does seem to cause problems.

3. Make the FATs

On a freshly formatted disk a FAT is one sector that contains $f7ffff00 followed by 508 x $00. A disk actually contains two FATs the second one being an exact duplicate of the first (supposedly in case the first gets damaged). To make our FATs we firstly prototype them in memory:

```
move.l    #buffer,a0        ; memory address
move.l    #$f7ffff00,(a0)+  ; FAT header
```

```
    move.l    #126,d0         ; and 508 x $00
    clr.l     d1
loop: move.l  d1,(a0)+
    dbra      d0,loop
```

Then using the rwabs call we write it to the disk. The rwabs call requires us to pass as a parameter the logical sector to start writing to. The bootsector which is the first sector on the disk is logical sector 0. The FATs occur at logical sectors 1 and 6.

```
* EX11.S
* This program formats a disk: single-sided only.

gemdos equ 1
bios equ 13
xbios equ 14

start:
    move.l    #str1,d0
    jsr       message
    jsr       key_press

    ; check key press

    cmpi.b    #$59,d0         ; Y
    beq       format
    cmpi.b    #$79,d0         ; y
    beq       format

quit:
    move.w    #$0,-(sp)       ; p_term
    trap      #gemdos         ; quit cleanly
    addq.l    #2,sp

format:
    move.l    #str2,d0
    jsr       message

    clr.l     d6
    move.w    #0,d7           ; track number to format
```

```
f_loop:
    move.w   d6,-(sp)      ; format code
    move.l   #$87654321,-(sp) ; magic number
    move.w   #1,-(sp)      ; interleave
    move.w   #0,-(sp)      ; side number
    move.w   d7,-(sp)      ; track number
    move.w   #9,-(sp)      ; sectors per track
    move.w   #0,-(sp)      ; drive number
    move.l   #0,-(sp)      ; reserved
    move.l   #buffer,-(sp) ; buffer address
    move.w   #$a,-(sp)     ; opcode - _flopfmt
    trap     #xbios
    add.l    #26,sp        ; tidy stack

    cmpi.w   #0,d0         ; check for errors
    bne      error         ; bne= branch if not equal

    addq.w   #1,d7         ; increase track number

    cmpi.w   #2,d7         ; if two tracks have been formatted
    bne      no_change     ;
    move.w   #$e5e5,d6     ; change the format code
no_change:

    cmpi.w   #80,d7        ; check to see if finished
    blt      f_loop        ; blt= branch if less than

    ; now make bootsector

    ; get random number

    move.w   #$11,-(sp)    ; _random
    trap     #xbios
    adda.l   #2,sp         ; tidy stack

    ; prototype boot sector

    move.w   #0,-(sp)      ; executable flag
    move.w   #2,-(sp)      ; disktype
    move.l   d0,-(sp)      ; serial number
    move.l   #buffer,-(sp) ; buffer address
```

```
	move.w	#$12,-(sp)	; _protobt
	trap	#xbios
	add.l	#14,sp	; tidy stack

	; and write it to the disk

	move.w	#1,-(sp)	; number of sectors to write
	move.w	#0,-(sp)	; side number
	move.w	#0,-(sp)	; track number
	move.w	#1,-(sp)	; start sector number
	move.w	#0,-(sp)	; drive number
	move.l	#0,-(sp)	; reserved
	move.l	#buffer,-(sp)	; buffer address
	move.w	#$9,-(sp)	; _flopwr
	trap	#xbios
	add.l	#20,sp	; tidy stack

	cmpi.w	#0,d0	; check for errors
	bne	error

	; make a FAT

	move.l	#buffer,a0	; buffer address
	move.l	#$f7ffff00,(a0)+	; FAT header
	move.l	#126,d0	; and clear the rest
	clr.l	d1
loop:	move.l	d1,(a0)+
	dbra	d0,loop

	; and write it to disk - twice

	move.w	#1,d0	; FAT #1
	jsr	sector_write
	cmpi.w	#0,d0	; check for errors
	bne	error

	move.w	#6,d0	; FAT #2
	jsr	sector_write
	cmpi.w	#0,d0	; check for errors
	bne	error
```

```
        move.l    #str4,d0     ; display finished message
        jsr       message
        jsr       key_press
        bra       quit

error:
        move.l    #str3,d0     ; display error message
        jsr       message
        jsr       key_press
        bra       quit

key_press:
        move.w    #$2,-(sp)      ; device - the keyboard
        move.w    #$2,-(sp)      ; bconin
        trap      #bios        ; read a character
        addq.l    #4,sp        ; tidy stack
        rts

message:
        move.l    d0,-(sp)     ; address of our string
        move.w    #$9,-(sp)    ; c_conws
        trap      #gemdos      ; write a string to the screen
        addq.l    #6,sp        ; tidy stack
        rts

sector_write:
        move.w    #0,-(sp)       ; drive number
        move.w    d0,-(sp)     ; start sector number
        move.w    #1,-(sp)     ; number of sectors to write
        move.l    #buffer,-(sp) ; buffer address
        move.w    #3,-(sp)      ; flag - write
        move.w    #$4,-(sp)      ; rwabs
        trap      #bios
        add.l     #14,sp       ; tidy stack
        rts

str1:   dc.b      "** Do you really want to format the disk in Drive"
        dc.b      " A? (Y)" or (N) ***",13,10,10,0
str2:   dc.b      "Formatting disk... ",13,10,0
str3:   dc.b      "An error has occurred during formatting. Press"
        dc.b      " any key to quit",0
```

```
str4:   dc.b    "Disc formatted successfully. Press any key to quit",0

        bss
buffer:         ds.w   5000
```

As this routine only formats a single sided disk, if you wish to convert it to format a disk double sided, you must bear in mind the following points.

1. Alternate sides when formatting each track.

```
format    side 0    track 0
          side 1    track 0
          side 0    track 1
          side 1    track 1
          etc.
```

2. When setting the format code (virgin) to 0 for the first two tracks remember that side 0 track 0 and side 1 track 0 are the first two.

Chapter 10
Introducing GEM

This chapter introduces the reader to GEM programming via assembly language; the GEM header, and other conventions.

To use the GEM libraries, ie the AES, and VDI, function calls in the same way that the previous chapter's used the BIOS etc, first a header or shell has to be set up so that we can properly use GEM. Like the BIOS etc, the GEM libraries consist of many ROM functions or routines that enable us to use these libraries for ourselves.

Resources

However, there is one caveat to using assembler when accessing GEM. Some of the routines to use menus, and dialog boxes are very involved and as there is resource construction kit on the disk we are fortunate in that we don't have to spend hours constructing menus, and complicated dialog boxes from scratch. So, although some details of constructing dialog boxes by hand (ie from the basics) are included this book does not go into the hand construction of drop down menus, as these are especially awkward to manage.

After constructing a few dialog boxes by hand you will soon appreciate the usefulness of a resource kit, and except for possibly including bit images in your dialog boxes, which the resource kit cannot handle, you may never want to construct them by hand again!

So in essence a resource construction kit enables the programmer to create drop down menus, and dialog boxes. To see an example of a dialog box see the *Assembly* options box in zzSoft's text editor, or see almost any GEM program. Once the dialog boxes, etc have been created in the resource kit a file containing all the information is saved in a file with the extension '.RSC'. The program that is going to use this resource file loads the resource file and allocates the information accordingly. This process will be looked at in more detail later on.

GEM header

So what is needed to utilise GEM in our programs. First a header and its associated user stack pointer has to be set up. This looks very complicated at first, and the reasons for doing this are also quite complex, but once the header (plus a few buffers, and other things) has been set up they can almost be ignored and programming can go ahead as usual.

The main reason for the header is that GEM allocates all the available memory to that program as it is invoked and if the program needs to use any calls that allocates memory then the program should de-allocate the memory it is not using at startup. This is done via the XBIOS 'Setblock' function and the header. At the same time a user stack is needed and this is added to the 'setblock' function. This procedure does not apply to desk accessories.

In practise the GEM header is invariably used in this form:

```
* GEM header and user stack
* header
        move.l    a7,a5        ; save a7
        move.l    #ustk,a7     ; stack pointer to our stack
        move.l    4(a5),a5     ; base page address
        move.l    12(a5),d0    ; base page offset to text length
        add.l     20(a5),d0    ; base page offset to data length
        add.l     28(a5),d0    ; base page offset to bss length
        add.l     #$100,d0     ; base page size
        move.l    d0,-(sp)
        move.l    a5,-(sp)
        move      d0,-(sp)     ; dummy value
        move      #$4a,-(sp)   ; 'setblock'
        trap      #1
        add.l     #12,sp

* initiate GEM application: 'appl_init'

* GEM program goes here

*end GEM application: 'appl_exit'

                  ds.l    256
```

```
ustk:       ds.l    1
```

In addition to the header and new stack, GEM expects the programmer to set up some space for it in the following arrays (or buffers):

```
contrl:     ds.w    12      ; control parameters
intin:      ds.w    128     ; input parameters
intout:     ds.w    128     ; output parameters
global:     ds.w    16      ; global parameters
addrin:     ds.w    128     ; input address
addrout:    ds.w    128     ; output address
```

These arrays are set up so that we can pass and receive information from GEM, with the information being passed by the programmer in this way:

```
* evnt_keybd
    move        #20,contrl      ; function number
    move        #0,contrl+2     ; number of integer inputs to intin
    move        #1,contrl+4     ; number of integer results from
* intout
    move        #0,contrl+6     ; number of input addresses passed
* to addrin
    move        #0,contrl+8     ; number of addresses returned by
*addrout

* call AES to operate function

* return code of key pressed in intout
```

This particular AES call is called 'evnt_keybd' and is similar to 'conin' in that it just waits for a key press, but the result is not passed to register d0, but via the 'intout' array.

Once again the question can be asked how do I know what parameters are needed to call the AES, and the VDI? All the necessary parameters are listed on disk in a similar manner to the BIOS calls.

In order to use GEM we need to initialise an application via the AES call 'appl_init', and once we have finished with the program we need to tell GEM we have finished with it by the use of 'appl_exit'. Although I have found that it is not always necessary to actually include these two calls in a GEM program it is always wise to follow the proper programming procedures as recommended by the people who wrote GEM.

Also, when calling GEM via the AES it is necessary to set up the AES parameter block, which contains the addresses of the six data arrays. They must be arranged in the following manner as this is how GEM expects to find them.

```
aespb:  dc.l    contrl,global,intin,intout,addrin,addrout
```

To call the AES the AES parameter block is placed in register d1, and then the AES identification code, #$c8, is passed to register d0, and then trap #2 is called. This is done like this:

```
* call the AES
    move.l      #aespb,d1
    move.l      #$c8,d0
    trap        #2
```

We can now put it all together to form the first GEM program:

```
* GEM1.S
* This simple GEM program just waits for a key press. Although
* simple it shows the basic outline or shell of a GEM AES program.

* header
    move.l      a7,a5
    move.l      #ustk,a7
    move.l      4(a5),a5
    move.l      12(a5),d0
    add.l       20(a5),d0
    add.l       28(a5),d0
    add.l       #$100,d0
    move.l      d0,-(sp)
    move.l      a5,-(sp)
    move        d0,-(sp)
    move        #$4a,-(sp)
    trap        #1
    add.l       #12,sp
```

```
* appl_intit
    move        #10,contrl    ; function number or opcode
    move        #0,contrl+2
    move        #1,contrl+4
    move        #0,contrl+6
    move        #0,contrl+8
    jsr         aes           ; call AES

* evnt_keybd  (wait for key press)
    move        #20,contrl
    move        #0,contrl+2
    move        #1,contrl+4
    move        #0,contrl+6
    move        #0,contrl+8
    jsr         aes           ; call AES

* appl_exit
    move        #19,contrl
    move        #0,contrl+2
    move        #1,contrl+4
    move        #0,contrl+6
    move        #0,contrl+8
    jsr         aes           ; call AES

* pterm -exit cleanly
    move        #10,-(sp)
    move        #$4c,-(sp)
    trap        #1

* AES subroutine
aes: move.l     #aespb,d1
    move.l      #$c8,d0
    trap        #2
    rts

        ds.l    256
ustk:   ds.l    1

* GEM arrays
contrl:         ds.w    12
```

```
intin:      ds.w    128
intout:     ds.w    128
global:     ds.w    16
addrin:     ds.w    128
addrout:    ds.w    128

aespb:      dc.l    contrl,global,intin,intout,addrin,addrout
```

Although this program is simple enough it does show the basic setup of a GEM program. To utilise the VDI other arrays and parameters have to be included too, but thus will have to wait for another chapter.

Please note the GEM arrays are user definable in that the amount defined for each array should be determined by the programmer. However, the amounts set above for the arrays should be sufficient for most purposes. This also applies to the user stack amount. Too little and the program will not function correctly.

GEM1.S can be written in another way, and it is perhaps more usual to see the 'contrl' parameters passed this way.

```
* GEM2.S

* header
    move.l      a7,a5
    move.l      #ustk,a7
    move.l      4(a5),a5
    move.l      12(a5),d0
    add.l       20(a5),d0
    add.l       28(a5),d0
    add.l       #$100,d0
    move.l      d0,-(sp)
    move.l      a5,-(sp)
    move        d0,-(sp)
    move        #$4a,-(sp)
    trap        #1
    add.l       #12,sp

* appl_init
    move.l      #appl_init,aespb
```

```
      jsr          aes       ; call AES
* evnt_keybd  (wait for key press)
      move.l       #evnt_keybd,aespb
      jsr          aes       ; call AES
* appl_exit
      move.l       #appl_exit,aespb
      jsr          aes       ; call AES
* pterm -exit cleanly
      move         #10,-(sp)
      move         #$4c,-(sp)
      trap         #1


* AES subroutine

aes:  move.l       #aespb,d1
      move.l       #$c8,d0
      trap         #2
      rts

      ds.l   256
ustk: ds.l   1

* GEM arrays
contrl:            ds.w   12
intin:             ds.w   128
intout:            ds.w   128
global:            ds.w   16
addrin:            ds.w   128
addrout:           ds.w   128

aespb:             dc.l   contrl,global,intin,intout,addrin,addrout
appl_init:         dc.w   10,0,1,0,0
appl_exit:         dc.w   19,0,1,0,0
evnt_keybd:        dc.w   20,0,1,0,0
```

In the first example the 'contrl' parameters were passed directly to 'contrl', but if the AES parameter block is examined it can be seen that the 'contrl' array comes first so by moving the address of each function each parameter can be passed directly to the AES parameter block. If any other parameters have to be passed, eg to 'intin' then these would have to be passed separately.

Please note that VDI, AES, BIOS, XBIOS, and GEMDOS calls may be freely used in the same source code although some care is needed in practise when mixing similar AES and VDI calls. In the above example AES calls and a GEMDOS call, 'pterm', are used together in the same program.

Chapter 11
Introducing the VDI

This chapter looks at a simple GEM VDI program, and also examines the 'virtual' and 'physical' work station concepts.

```
* GEM3.S This program uses the VDI call vq_mouse() which waits
* for a right mouse button press. GDOS is also checked for.

gemdos   equ  1

* GEM header
     move.l      a7,a5
     move.l      #ustk,a7
     move.l      4(a5),a5
     move.l      12(a5),d0
     add.l       20(a5),d0
     add.l       28(a5),d0
     add.l       #$100,d0
     move.l      d0,-(sp)
     move.l      a5,-(sp)
     move        d0,-(sp)
     move        #$4a,-(sp)
     trap        #gemdos
     add.l       #12,sp

* get current screen res
     move        #4,-(sp)
     trap        #14
     addq.l      #2,sp
* res returned in d0
     move         d0,res      ; store screen resolution
* appl_init()
     move.l       #appl_init,aespb
     jsr          aes         ; call AES

* graf_handle()
     move.l       #graf_handle,aespb    ; get physical screen handle
```

```
    jsr        aes
    move       intout,gr_handle     ; store handle

* start by opening a virtual workstation
    move       #100,contrl
    move       #0,contrl+2
    move       #11,contrl+6

* is GDOS present
    moveq      #-2,d0
    trap       #2
    addq       #2,d0
    beq        no_gdos      ; no GDOS
    move       res,d0
    add        #2,d0
    move       d0,intin

    bra        s_no_gdos
no_gdos:

    move       #1,intin              ; default if GDOS not
* loaded
s_no_gdos:
    move       #1,intin+2    ; line type
    move       #1,intin+4    ; colour for line
    move       #1,intin+6    ; type of marking
    move       #1,intin+8    ; colour of marking
    move       #1,intin+10   ; character set
    move       #1,intin+12   ; text colour
    move       #1,intin+14   ; fill type
    move       #1,intin+16   ; fill pattern index
    move       #1,intin+18   ; fill colour
    move       #2,intin+20   ; coordinate flag
    move.w     gr_handle,contrl+12 ; device handle
    jsr        vdi          ; v_opnvwk open virtual work station
    move.w     contrl+12,ws_handle  ; store virtual workstation handle

* ///////////////////////////////////// program goes here
* sample mouse button state: vq_mouse()
sample_again:
    move       #124,contrl
```

```
        move.w      #0,contrl+2
        move.w      #0,contrl+6
        move.w      ws_handle,contrl+12
        jsr         vdi
        cmpi.w      #2,intout       ; right mouse button
        bne         sample_again
* //////////////////////////////// end of program

* exit

* close the virtual workstation
* v_clsvwk()
        move        #101,contrl
        clr.w       contrl+2
        clr.w       contrl+6
        move.w      ws_handle,contrl+12
        jsr         vdi

* appl_exit()
        move.l      #appl_exit,aespb
        jsr         aes     ; call AES

* pterm -exit cleanly
        move        #10,-(sp)
        move        #$4c,-(sp)
        trap        #gemdos


* AES subroutine

aes:    movem.l     d0-d7/a0-a6,-(sp)
        move.l      #aespb,d1
        move.l      #$c8,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

vdi:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #vdipb,d1
        moveq.l     #$73,d0
```

```
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

            ds.l    256
ustk:       ds.l    1

* GEM arrays
contrl:         ds.w    128
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    16
addrin:         ds.w    128
addrout:        ds.w    128

* for vdi
ptsout:         ds.w    128
ptsin:          ds.w    128

vdipb:  dc.l    contrl,intin,ptsin,intout,ptsout

aespb:          dc.l    contrl,global,intin,intout,addrin,addrout
appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
graf_handle:    dc.w    77,0,5,0,0

gr_handle:      dc.w    1
res:            ds.w    1
ws_handle:      ds.w    1
```

GEM divides its output world into 'virtual' and 'physical' workstations or devices. In practise GEM opens the screen as a physical workstation for us, whilst we have to open up any other physical devices for ourselves. In practise this usually means directing the output to a printer which GEM, and specifically the VDI would refer to as a physical work-station.

Virtual workstation

The desktop, user programs and desktop accessories all have to use the screen, and so that each application can use the VDI without affecting the other application, we have to allocate ourselves a virtual screen workstation. GEM uses the word 'virtual' to mean 'as if' or 'pseudo' device. Each virtual workstation opened can be then directed to the screen without affecting any other graphic settings. In this book and often in practise only one virtual workstation is opened and one physical workstation is opened: a printer.

Because the VDI can send its output to a variety of devices most usually the screen, plotter, printer and metafile, each workstation is given a handle so that the output can be sent to that device by reference to that handle. So if a printer workstation was opened its handle – which is a number allocated to that device – would be used each time the output of the VDI was wanted to be sent to the printer. A practical example of opening a physical workstation– a printer will be given at a later stage.

GDOS

Note to open a physical workstation GDOS needs to have been loaded. GDOS is an acronym for Graphics Device Operating System and was left out of the GEM ROMS, so it has to loaded separately. Invariably it is loaded via an AUTO folder at boot up. Failure to boot with GDOS will crash the ST without warning when attempting to open a physical workstation.

An example VDI call:

```
* sample mouse button state: vq_mouse()
    move      #124,contrl     ; function opcode (number)
    move.w    #0,contrl+2     ; number of coordinate points in
* ptsin array
    move.w    #0,contrl+6     ; number of input parameters in
* intin array
    move.w    ws_handle,contrl+12 ; device handle
    jsr       vdi
```

Besides these parameters others involved could be:

```
    contrl+4   ; number of coordinate points in ptsout array
    contrl+8   ; number of output parameters in intout array
    contrl+12  ; sub function number
```

In various GEM technical manuals you may see the contrl array elements referred to as contrl(0), contrl(1), contrl(2), etc, where contrl(0)= contrl, contrl(2)= contrl+2, and contrl(2)= contrl+4 in assembly language. This is because the 'contrl' array is accessed by word length parameters.

graf_handle

When using the VDI in our program the first thing we need to do is to get the physical screen handle via the 'graf_handle' call. This handle is then passed to the virtual workstation and from this we get the virtual workstation handle which is then used for all further VDI calls to the screen. If any other physical workstations are opened (eg a printer) then we have to get the handle for that device so that any output can be accessed via its handle.

To call the VDI the address of the VDI parameter block is placed in d1, and the VDI code, #$78 is placed in d0. This is obviously very similar to the AES sequence.

GDOS again

When a virtual workstation is opened it is necessary to first determine whether GDOS has been loaded. This is done with the call

```
moveq       #-2,d0
trap        #2
addq        #2,d0
```

If d0 is equal to 0 after the trap then GDOS has not been loaded then a one must be passed to the intin array, as the first word of intin expects the device driver identification (see ASSIGN.SYS later). IF GDOS is present then two should be added to the result and this should be passed to the intin array. By passing 1's to the rest of the intin array the default GEM values will be used by the program for the various VDI graphic operations.

NCD or raster coordinates

However, 'intin+20' needs to be passed a suitable value as this determines the coordinates used by GEM. There are two possibilities here: a one passed to 'intin+20' would tell GEM that you wanted NDC (Normalized Device Coordinates) coordinates to be used in graphic output to the screen. A two tells GEM that raster coordinates should be used, which is the coordinate system usually used. NCD uses 32,768 pixels by 32,768 pixels but as there is no output device that can handle this type of resolution raster coordinates are usually used where 0,0 indicates the x and y coordinates respectively starting at the top left of the screen. Each point (pixel) that can be plotted on the display screen is represented by the raster coordinate system, where the actual dimensions are governed by the screen resolution. In high res the x coordinate goes from 0 to 639, whilst the y coordinate goes from 0 to 399. In medium res the y coordinate goes from 0 to 199, and in low res the x coordinate goes from 0 to 319 with the y coordinate from 0 to 199.

Workstation capabilities

Once the virtual workstation has been opened the output array lists a variety of pertinent information about the screen and what it can support. More information about this can be found from the 'vq_extnd' call which lists further information about the graphic capabilities of the workstation eg text alignment, colour information, etc. See disk for full coverage of the information available from these GEM calls. Suffice it to say that the screen supports all the VDI graphic calls in the VDI library as listed on the disk.

vq_mouse

Next the VDI 'vq_mouse' call is made which waits for the user to press the right mouse button. Pressing the left button or keyboard results in the call being operated again by the 'sample_again' loop until the right mouse button is pressed.

Exiting the workstation

To exit, the virtual workstation(s) and any other physical workstation must be closed and then the 'appl_exit' call should be made (in that order), and finally 'pterm' so that we exit back correctly to the calling program.

Note that GEM3.S is set up for AES and VDI calls. If you only want to call the VDI the AES stuff can be left out.

VDI printer output

One important feature of the VDI graphic output, including text, is that it is possible to ensure that the output is at the resolution of the device. In practise this means that printing text on the the high resolution screen at 90*90 dots per inch (dpi) results in text being printed to a 9–pin dot matrix printer at 120*144 dpi, to 24–pin printer at 180*180 dpi or even 360*360 dpi. If a screen dump was sent to the printer the output resolution would be similar to the screen, and therefore not nearly as good. This is covered in very much more detail later on.

Chapter 12
GEM Objects

This chapter examines the data structures of GEM objects that make up the construction of dialog boxes and a simple dialog box is constructed from first principles.

Constructing a dialog box

GEM refers to the parts that make up a dialog box, or alert box, or drop down menu as objects. Each object has a special name and function most of which are available in the Resource Construction Program (RCP). However, in this chapter we are going to construct a simple dialog box by hand ie from first principles, but to do this we have to examine the basic structures that make up a GEM object.

First object types are examined, by which is meant the basic types of boxes, text, icons, and bit images that are available to the programmer when constructing a dialog box or menu.

Object types

The following are the type of objects available:

20 g_box
21 g_text
22 g_boxtext
23 g_image
24 g_progdef
25 g_ibox
26 g_button
27 g_boxchar
28 g_string
29 g_ftext
30 g_fboxtext
31 g_icon
32 g_title

g_box a rectangular box with an optional border

g_text a text string, which can have various characteristics. Uses 'tedinfo' structure.

g_boxtext a rectangular box that contains text, as g__text.

g_image a mono only bit image that points to 'bitblk' structure.

g_progdef an object defined by programmer; uses 'applblk' structure.

g_ibox an invisible rectangle usually used to group together other objects, often radio buttons.

g_button centred text in default font in a rectangle, usually used as a radio button.

g_boxchar as above but just one character allowed.

g_string a string in the default font.

gftext a formatted text string that can be edited. Uses 'tedinfo' structure.

g_fboxtext as above but contained within a rectangle.

g_icon a mono image with mask (icon). Uses 'iconblk' structure.

g_title a special 'g__string' for use in GEM menu bar titles.

The RCP does not allow the use of *g_image*, *g_progdef*, and *g_icon* objects. For more information on 'tedinfo' structure see later. Using 'g__image' from first principles is looked at in chapter sixteen.

Tree

In a dialog box there can be many GEM objects such as a 'g__string' which could be used to give the dialog box a title, or 'g__boxtext' to give it a title in a box; radio buttons; an editable text object so that user data can be entered for example changing a drive designation, say from A:\ to B:\, with the whole lot contained within a 'g__box'. The structure these various objects are grouped in is called a tree. Each object branches out from a parent (g__box) to other objects that themselves can be parents to other objects (children), whilst objects that have a common parent are called siblings. The actual arrangements of objects in a tree is quite complicated and as we have a RCP that does all this arranging for us no further theory will be discussed except to explain various concepts as they come about in the actual practise of constructing GEM objects.

Each object is defined by a 24 byte (12 word) list, and it is organized in this way:

Object structure:

Word	description
0	next object; index of child that is not first or last. If root, −1 (hex FFFF)
1	starting object; index of first child object
2	ending object; index of last child object
3	object type, eg g__box, g__button
4	object flags; selectability of object, see below
5	object status; state of object, see below
6 & 7	object specification; pointer to object data structure, eg tedinfo, or colour & thickness of box.
8	object x coordinate, relative to parent
9	object y coordinate, relative to parent
10	object width
11	object height

There is even more to come! Now can you see why using a RCP that sorts all this lot out for you has very distinct advantages!

The tedinfo data structure:

This structure is arranged as follows:

Word		description
0 & 1	te_ptext	pointer to actual string
2 & 3	te_ptmplt	pointer to format template
4 & 5	te_pvalid	pointer to validation string, see below
6	te_font	font size(3=normal, 5=small)
7	te_resvd1	reserved word (0)
8	te_just	text justification (0=left 1=right 2=centred).
9	te_colour	colour, see below
10	te_resvd2	reserved word (0)
11	te_thickness	thickness of rectangle, 0=no border, 1−128= thickness of inside border, −1 to −128 thickness of outside border.
12	te_txtlen	length of 'te_ptext' string+1
13	te_tmplen	length of 'te_ptmplt' string+1

te_pvalid

Validation code	characters allowed
9	digits 0−9
A	upper case letters (A to Z) or spaces
a	upper and lower letters and spaces
N	9+A
n	9+a
F	valid TOS filename chars including ? : ´
p	F+\
P	valid TOS filename chars including \ and:
X	All

Colour

Colour coding:

white	0	light grey	8
black	1	dark grey	9
red	2	light red	10
green	3	light green	11
blue	4	light blue	12
cyan	5	light cyan	13
yellow	6	light yellow	14
magenta	7	light magenta	15

te_color and object specification (for g_box, g_ibox, g_boxchar)

To select the border and text colour, write mode, fill pattern, and fill colour used in an object a bit arrangement is used:

	msb															lsb
bit	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

bit	
0–3	fill colour
4–6	fill pattern– 0=no fill, 1–6 dithered, 7 solid
7	writing mode: 0=transparent, 1=replace.
8–11	text colour
12–15	border colour

An additional word is used for the 2 word object specification, where bits 16–23 describe the border thickness, and bits 24–31 describe the ASCII value of 'g_boxchar' character.

Values for bits 16–23 are the same as 'te_thickness' in the tedinfo data structure.

Object flags

bit if set

0 selectable
1 default
2 exit
3 editable
4 rbutton
5 lastob
6 touchexit
7 hidetree
8 indirect

selectable: the user can select the object which then appears in reverse.

default: as above but can also be selected with the return key. Only sensible to have one object designated 'default'. Often used with the 'OK' button. When setting this bit you should make the rectangle holding the text thicker so that it stands out and the user can see that it is the default exit button.

exit: this allows the control of the dialog box to finish and return to the rest of the program. 'exit' would be used with 'selectable', and 'default' for an 'OK' button that would end the use of a dialog box.

editable: text held by the object can be entered/edited by the user.

rbutton: this stands for a radio button, which is a group of buttons usually arranged within an invisible box from which only one can be selected. When one button is selected any other choice is de-selected. Radio buttons must all be children of the same parent object.

lastob: this bit is set to show that this is the last object in the particular tree.

touchexit: as soon as the mouse pointer is over the object and the mouse button is pressed control is passed back to the calling program.

hidetree: all objects are made invisible to 'obj__draw' and 'obj__find'.

indirect: object points to another value.

Object status

bit if set

0 selected
1 crossed
2 checked
3 disabled
4 outlined
5 shadowed

selected: the object is displayed in reverse video to show that it has been selected.

crossed: the object has an 'X' drawn in the box.

checked: a tick appears in the box, or menu item.

disabled: text is greyed out.

outlined: a border is drawn around the object.

shadowed: a shadow falling to the lower right is drawn around the object.

```
* GEM4S This example shows the construction of a simple dialog
* box by hand, and how to display it on screen.

gemdos  equ 1

* header
        move.l  a7,a5
        move.l  #ustk,a7
        move.l  4(a5),a5
        move.l  12(a5),d0
```

```
    add.l       20(a5),d0
    add.l       28(a5),d0
    add.l       #$100,d0
    move.l      d0,-(sp)
    move.l      a5,-(sp)
    move        d0,-(sp)
    move        #$4a,-(sp)
    trap        #gemdos
    add.l       #12,sp

    bsr         form_cent     ; get centred coordinates
    bsr         obdraw        ; put dialog box on screen
    bsr         f_do          ; handle interaction

* pterm -exit cleanly
    move        #10,-(sp)
    move        #$4c,-(sp)
    trap        #gemdos

form_cent:
    move.l      #form_center,aespb      ; get coords of centred tree
    move.l      #parent,addrin
    bsr         aes
    movem.w     intout+2,d0-d3          ; returned in intout+2
    rts

obdraw:
    move        #0,intin       ; index of first object
    move        #1,intin+2     ; depth
    move        d0,intin+4     ; x coord
    move        d1,intin+6     ; y coord
    move        d2,intin+8     ; width
    move        d3,intin+10    ; height
    move.l      #parent,addrin ; address of parent dialog box tree
    move.l      #object_draw,aespb
    bsr         aes
    rts

f_do:
    move.l      #form_do,aespb
    clr.w       intin          ; No editable text field
```

```
    move.l      #parent,addrin
    bsr         aes
    rts

* AES subroutine
aes:
    move.l      #aespb,d1
    move.l      #$c8,d0
    trap        #2
    rts

                ds.l   256
ustk:           ds.l   1

text1:          dc.b   '     -----EXAMPLE----',0

text2:          dc.l   texty,textt2,textt2
                dc.w   3,0,2,$11f0,0,3,5,0

texty:          dc.b   'Exit',0

textt2:         dc.b   0

* GEM arrays

contrl:         ds.w   12
intin:          ds.w   128
intout:         ds.w   128
global:         ds.w   16
addrin:         ds.w   128
addrout:        ds.w   128

aespb:          dc.l   contrl,global,intin,intout,addrin,addrout

form_center:    dc.w   54,0,5,1,0
object_draw:    dc.w   42,6,1,1,0
form_do:        dc.w   50,1,2,1,0

* dialog box tree
parent:
    dc.w        -1,1,2,20,0,16        ; g_box
```

```
    dc.l      $00021100
    dc.w      170,100,250,100

    dc.w      2,-1,-1,28,0,0       ; g_string, title string
    dc.l      text1
    dc.w      10,10,5,1

    dc.w      0,-1,-1,22,7+32,0       ; g_boxtext, boxed exit button
    dc.l      text2
    dc.w      50,60,60,25
```

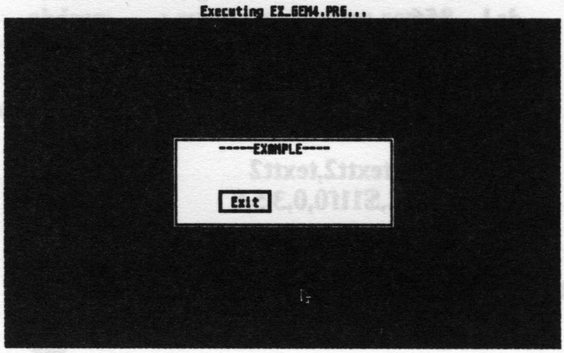

diagram 12:1 the result

To actually display a dialog box on screen GEM calls to the AES need to be made. The first is to 'form_center' which usefully returns the centred coordinates of the dialog box so that it can be centred on the screen. The second is to 'objc_draw' which draws the objects on screen and the next is to 'form_do' which handles the interaction between user and objects until we exit from the dialog box back to the program.

Looking in more detail at the program:

First the GEM header is set up with the user stack. Note that the stack buffer space is allocated above the 'ustk' label as the stack grows upward in memory.

Next 'form_center' an AES call is made. This returns the centred coordinates of the dialog tree in 'intout' so that it can be displayed in the centre of the screen by 'obj__draw'. If these returned coordinates were not used then we would have to calculate the coordinates ourselves. Obviously 'form__center' is very useful. However, if you wanted to place the dialog box anywhere on the screen entering the required coordinates into the 'objc__draw' 'intin' array would do the job. The first word of 'intout' is a reserved word so we need to get the coordinates from the second word, 'intout+2'.

The coordinates are returned in the following manner:

intout+2 centred x coord of tree
intout+4 centred y coord of tree
intout+6 width of tree
intout+8 height of tree

These results are placed in d0–d3 with the 'movem.w' instruction which places each sequential word held by 'intout' in each data register one by one.

To actually draw the dialog box on screen the AES call 'objc__draw' is next made using the coordinates returned by 'form__center' by placing them in the 'intin' array from 'intin+4'. This is because 'objc__draw' expects the number of the object (index) to be drawn first in the first word of 'intin' and the number of levels to be drawn in the second word. Invariably the first object would be zero in the first word. A value of seven in the second word would ensure that all (possible) levels of the dialog tree would be drawn.

So that the user can interact with the dialog box now on screen, the AES call 'form__do' needs to be invoked. This allows the mouse to be used to select any radio buttons or other types of button, move any sliders, or the user to edit or enter text from the keyboard. The first word of the 'intin' array should contain the number (index) of the first text field to be edited, but if there are no text fields this should be set to zero. This can be done with the instruction 'clr intin', which has the same result as 'move #0,intin', ie the first word of the 'intin' array will contain nothing.

'intout' contains the index of the object which was selected by the user to end interaction with the dialog box. In this case the 'exit' button, which can be selected by the mouse pointer and pressing the left mouse button or by pressing the 'Return' key.

Examining the first part of the dialog box tree we can see that it is arranged so that there are 6 words, 1 long, and another 4 words of data contained in it, which satisfies the conditions of the object structure.

```
* dialog box tree
parent:

        dc.w        -1,1,2,20,0,16        ; parent: g_box
        dc.l        $00021100
```

If we examine the first line of data:

```
        dc.w        -1,1,2,20,0,16        ; parent: g_box
```

This corresponds with the first 6 words of the object structure as defined earlier:

Word	description
0	next object; index of child that is not first or last. If root, −1 (hex FFFF):
1	starting object; index of first child object
2	ending object; index of last child object
3	object type, eg g__box, g__button
4	object flags; selectability of object
5	object status; state of object

The first word has the value '−1' which shows that it is the root object. The root object refers to the object that holds all the other objects, ie its parent.

The second word has the value '1' which is the index of the first child which is the next object, which is the object that holds the data for the dialog title, '-----Example-----'.

The third word has the value '2' which states that the 3rd object (counting from 0) is the last, the 'exit' button.

The fourth word has the value '20' which refers to 'g_box' type of object. This is the actual box to hold the other objects.

The fifth word has the value '0' which means that the object cannot be selected. It would not be correct if the main box was selectable, ie it would turn black when the mouse pointer was clicked over it.

The sixth word has the value 16 which means that the box should have an outline around it which it does.

Examining the second line of data (word 6 and 7) – object specification:

```
dc.l    $00021100
```

This refers to the colour and thickness of the border of the object– the 'g_box', where '$00021100' means:

0=	white fill colour
0=	no fill, and transparent writing mode
1=	text colour– black
1=	border colour– black
2=	inside border thickness

The last and third line corresponds to the 8–11 words of the object structure:

```
dc.w        170,100,250,100
```

refers to the x coordinate, y coordinate, width and height of the 'g_box', where x refers to distance across the screen (0 to 639 on a mono monitor), and y refers to the distance down the screen (0 to 399 on a mono monitor). This corresponds to the number of pixels on the screen.

If we now look at the second data structure in the dialog box:

```
dc.w     2,-1,-1,28,0,0     ; g_string, title string
dc.l     text1
dc.w     10,10,5,1
```

This follows the pattern described above but with some differences.

The first three words describes the next, start and end objects as we have seen. So this states that number '2' is the next object in the tree, and '–1' states that there are no children, so there are no next or end objects.

The next three words tell us that the object type is a 'g_string' and that it cannot be selected and has no special status, which is what we need for text.

'text1' is a label that points to or refers to the text which we want printed:

```
text1: dc.b  '    -----EXAMPLE----',0
```

The actual text to be printed must always be followed by a null byte.

The next four words describe the x, y, width, and height coordinates of the object. However, it is important to realise that the coordinates of the children of an object– in this case child of the 'g_box'– are relative to the parent.

The last object– the 'exit' button:

```
dc.w     0,-1,-1,22,7+32,0      ; g_boxtext, boxed exit button
dc.l     text2
dc.w     50,60,60,25
```

is also similar in construction but has the following differences:

The first three words describes the next, start and end objects. So this states that number '0' is the next object in the tree (the last object must point back to the parent). '–1' states that there are no children, so there are no next or end objects.

The next word value is '22' and describes a 'g_boxtext' which has its object flags (flag is a computer term to describe a particular state) set to 7 ie it is 'selectable', a 'default', and an 'exit' object which is ideal for an exit button. But what about '32'? As this is the last object in the tree, bit 5 is set so 32 is added to the object flag.

The next two words are an address pointer that refers to a tedinfo structure:

```
text2:   dc.l  texty,textt2,textt2
         dc.w  3,0,2,$11f0,0,3,5,0

texty:   dc.b  'Exit',0

textt2:  dc.b  0
```

Referring back to the tedinfo structure you will see that the first long word is 'te_ptext' a pointer to the string to be actually printed, in this case 'texty' which has defined 'Exit' to be the string.

The next two long words are not relevant to us as the text is not editable so they both point to a null 'textt2'.

The next 8 words refer to the font (5=normal), 0 for reserved word, 2= centred text, $11f0 for colour, 0 for reserved word, 3 for thickness of box, 5 for length of text+1 for null, and 0 for 'te_ptmplt' as it is not applicable.

'form_do' returns in the first word of the intout array the index of the object that caused 'form_do' to finish. In this case intout would contain '2'.

And that's it at last!

Chapter 13
Using MKRSC.PRG

This chapter takes a detailed look at using the supplied Resource Construction Program (RCP). With a step-by-step description of constructing a dialog box and a program to use it in assembly language the reader is taken further along the road to proficient GEM programming.

A resource construction program (RCP) allows the ST application programmer to design a useful and user-friendly program interface with the minimum of fuss. However to correctly use a RCP it is necessary to understand at least some of the basic theory underlying their construction and method of design and the reader is referred back to chapter twelve for reference.

The RCP on the disk can only be used with the resource files created by the program. Other resource files (created for example by WERCS) cannot (usually) be edited and altered within this RCP. The other limitation to this RCP is that bit images cannot be used in dialog boxes as the program does not support this. Fortunately it is fairly easy to put bit images in simple dialog boxes by creating the dialog box by hand. See chapter sixteen for more details of this.

Using MKRSC.PRG

To use the RCP it should be double clicked from the desktop or run from the text editor by selecting the *Run Other* option from the *Program* drop down menu.

Constructing a dialog box

Before reading the next part of the chapter it would be as well to run GEM5.PRG and look at the finished dialog box that is about to be constructed. Diagram 13:1 shows the finished dialog box within the RCP.

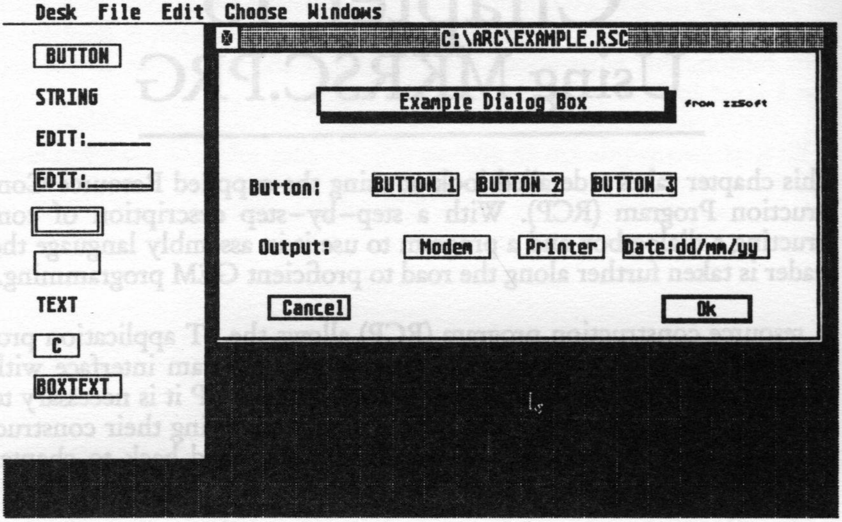

diagram 13:1

The dialog box consists of three radio buttons in the centre of the box, called BUTTON 1, BUTTON 2, and BUTTON 3, with text on the left describing them, entitled 'Button:'. The next row entitled 'Output' holds two radio buttons which hold the legends 'Modem' and 'Printer'. The next button is an editable button and it holds the date to be entered by the user: dd/mm/yy, where this refers to the day (dd), month (mm), and year (yy). The bottom two buttons are the usual 'Cancel' and 'Ok' objects. Notice that the 'Ok' button has a thicker border indicating it is the default object and will be selected by pressing the Return key. The whole dialog box is described at the top by a shadowed box with the title in it: 'Example Dialog Box'. Finally to the right of this is the text 'from zzSoft' in small text.

Note that the dialog is an example only and is used only as such.

As discussed in the previous chapter GEM refers to the parts that make up a dialog box, or alert box, or drop down menu as objects. Each object has a special name and function most of which are available in the RCP as icons. When the RCP is first run we are presented with three icons to the left of the screen: 'Menu', 'dialog', and 'unknown', menus and dialog boxes being the two most useful types. To make a start New should now be selected from the File drop down menu and the screen should change to diagram 13:2 which shows a window opened for ready for use by the programmer. The dialog box icon should now be selected and dragged over to the window– see diagram 13:3, when the name of the tree (dialog box) under construction is presented to us to alter or agree with. I usually leave it as it is and select 'Ok'.

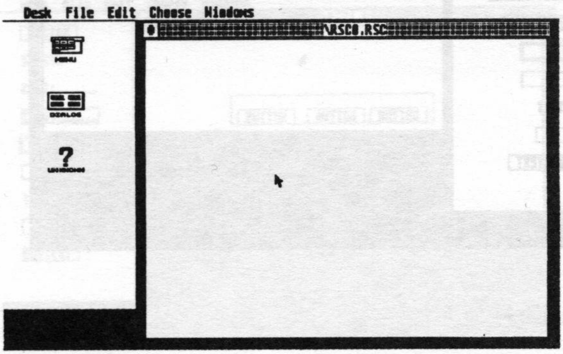

diagram 13:2

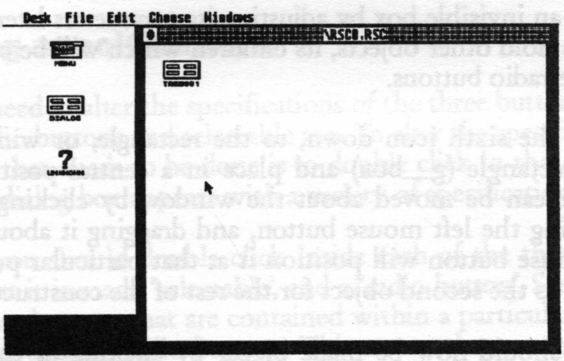

diagram 13:3

Next the dialog box icon should be double clicked with left mouse button pointer, and the screen should alter to diagram 13:4 which shows a object type 'g_box'– a plain box with a border. The size of this rectangle can be altered by dragging the bottom left–hand corner either in or out to make it smaller or bigger respectively. However, I have chosen to leave it at the default size.

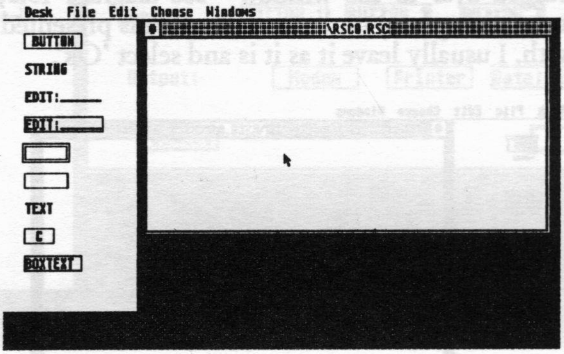

diagram 13:4

The left–hand side of the screen shows the types of GEM objects that can be used in our dialog box, shown in icon form. Obviously an invisible box cannot be shown so I have assumed that the sixth icon down can be used as an invisible box by adjusting its parameters later on. This will be used to hold other objects, its children which will be configured to be selectable radio buttons.

So, next drag, the sixth icon down, to the rectangle, or window that contains the rectangle (g_box) and place in a central position. This small rectangle can be moved about the window by clicking inside it without releasing the left mouse button, and dragging it about. Releasing the left mouse button will position it at that particular point. This will be known as the second object for the rest of the construction.

The rectangle should now be made bigger by clicking in the bottom right–hand corner and dragging the outline of the rectangle until it is the required size to hold three radio buttons. If you get the size wrong it is very easy to correct in a similar manner.

Next the first icon– rectangle with 'button' in it should be dragged into the second object. This should be done three times until the three buttons are situated symmetrically in the second rectangle. See diagram 13:5. Note that by moving the second object about the three buttons contained within it are also moved with it, this is because the three buttons are now children of the second object, whilst the second object is a child of the first, large object which holds them all.

diagram 13:5

Altering specifications

We now need to alter the specifications of the three buttons so that they can be radio buttons and selectable too. To alter the specifications of any object all that needs to be done is to double click in that object. Immediately a dialog box appears with a variety of specification choices.

So, next you should double click inside each of the three buttons and ensure that it is made 'selectable', and a 'radio button'. See diagram 13:6. Only those buttons that are contained within a particular parent object can be (dependant) radio buttons. This means that another set of radio buttons can be created within another parent object without affecting the integrity of any other radio button grouping.

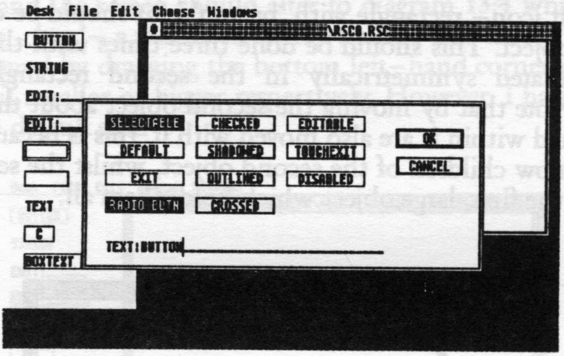

diagram 13:6

Once the second object is positioned correctly then this may be double–clicked and the border set to zero so that it is an invisible box. This does not affect the objects relationship with its children or parent. It should not be made selectable etc. Note that this box does not have to made invisible, it is only done this way as it looks better. It is perfectly all right to leave the parent box visible.

Naming the objects

The three radio buttons should then be selected one by one so that they can be named. When an object is named it should be named so that is easy to identify later on. For instance when a button is the 'Ok' button then it should be named 'OK', and so on. The names we give to the objects should not be confused with the text contained in the object. We give each object a name because when the resource file is saved a file with the extension .H (H= Header) is saved with the same name as the resource file. In this file are the names that we have given to the objects. It is only sensible to name those objects that are selectable as those are the only ones we would be interested in. There is no particular point in naming a title or a non–editable string.

To name an object click over it so that it becomes selected– ie it goes black (known as reverse video). I found that some objects were difficult to select, but if I held the left shift key down at the same time as clicking over the objects they could be selected ok. Also holding down the control key at the same time as clicking on an object selects its parent which can be useful. Then select 'name' from the Choose menu.

My EXAMPLE.H looked like this:

```
#define    TREE001     0
#define    CANCEL      2
#define    BUTTON2     4
#define    BUTTON1     5
#define    BUTTON3     6
#define    DATE        9
#define    MODEM       11
#define    PRINTER     12
#define    OK          14
```

which I later altered to equates. See the example source code file, GEM5.S. It will become more apparent as we go on as to why naming objects can be so useful.

Further #define's are for tree002, used for a drop down menu– see next chapter.

To get the title object 'Example Dialog Box' is very easy. Drag the last object icon 'boxtext' to the top of the dialog box, and double click to alter its specifications. Enter the required text in the 'PTEXT' field, alter the size of the border, select shadowed. Ensure that it is not editable, and ignore the PTMPLT, and PVALID fields . That's it.

Next the the 'modem' and 'printer' radio buttons should be created by following the procedures as outlined above. Note that I named these objects for the header file the same as the text in the button.

Editable text

The date object is the next to be created by dragging the fourth icon object (EDIT:________) across to the dialog box and positioning it next to the printer button. Double–clicking on this object will show the dialog box as illustrated in diagram 13:7.

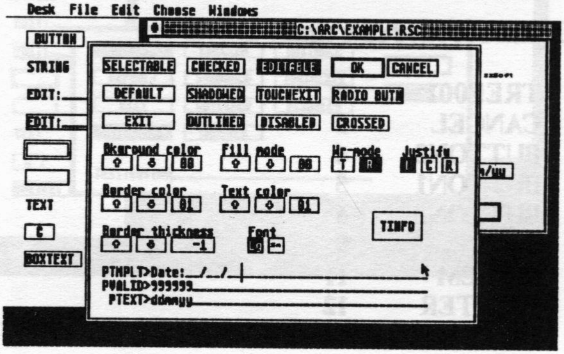

diagram 13:7

The 'Date' object demonstrates using editable text with a template, and a validation string. The form it takes is:

PTMPLT>date:____/____/____
PVALID>999999
PTEXT>ddmmyy

From the previous chapter these will be remembered from the tedinfo data structure, except each name was proceeded by 'te_', eg 'te_ptmplt'. 'PTMPLT' is the template the object's text takes, whilst the validation string only allows digits which is appropriate for a date to be entered in this format– pressing any key other than a number key will result in no action being taken by 'form_do'. The actual text initially output is 'ddmmyy'. Note that it is possible to pass the actual date taken from the computer, but this would have to done from assembly language and not from the RCP.

Editing text

A dialog box allows the input and editing of text (see next two chapter's, and their associated programs for more details on dialog boxes with editable text fields) where the objects are specified as editable. To make editing easy GEM allows the following functions:

Escape: All characters are erased from the field

Return or Enter: If an object has the flag 'default' then this is selected by GEM and the dialog box is ended immediately and control passed from GEM ('form_do' or 'objc_edit') to the application.

Backspace: the character to the left of the cursor is deleted and the cursor is moved one position to the left

Delete: deletes the character to the right of the cursor

Up arrow or Shift+Tab: the cursor is moved to the previous editable object, and positioned at the next writing position.

Down arrow or Tab: As above but moved to next input field.

Left and right arrows: the cursor is moved over text to the left or right.

The 'Ok' and 'Cancel' buttons are constructed from the first icon, 'button'. They are usually both given an 'exit', and selectable spec, whilst 'Ok' is made 'default'. They should be named appropriately as I have done.

The resource file should now be saved with an appropriate name– I called mine 'example.rsc'. Note that the resource file should be in the root directory, ie it should not be placed in a folder even if the calling program is executed from one, unless 'rsc_load' is passed the folder path in your program.

The dialog box can now be displayed on then screen in the following manner:

```
* GEM5.S
* Load and display an example Resource file: EXAMPLE.RSC
* Use AES calls only
* Equates modified from the file EXAMPLE.H
```

```
* NOTE: resource file must be in root directory of drive program run
* from.

tree001     equ   0
cancel      equ   2
button2     equ   4
button1     equ   5
button3     equ   6
date        equ   9
modem       equ   11
printer     equ   12
ok          equ   14

* header
      move.l   a7,a5
      move.l   #ustk,a7
      move.l   4(a5),a5
      move.l   12(a5),d0
      add.l    20(a5),d0
      add.l    28(a5),d0
      add.l    #$100,d0
      move.l   d0,-(sp)
      move.l   a5,-(sp)
      move     d0,-(sp)
      move     #$4a,-(sp)
      trap     #1
      add.l    #12,sp

      move.l   #rsc_load,aespb     ; AES load a resource file
      move.l   #rsc_file,addrin    ; name of resource file to be loaded
      jsr      aes
      cmpi.w   #0,intout           ; was the resource file loaded
      beq      exit                ; no

      move.l   #rsc_gaddr,aespb    ; get address of resource tree
      move     #0,intin            ; tree structure
      move     #0,intin+2
      bsr      aes
      cmpi.w   #0,intout           ; error
      beq   exit                   ; yes
```

```
    move.l    addrout,parent     ; place address in parent

    bsr       form_center        ; get centred coords of dialog box
    bsr       obdraw             ; draw it on screen
    bsr       f_do               ; handle interaction with user

    move.l    parent,a0

    add.l     #(printer*24)+10,a0    ; get address of object status -
* 'printer'

* test to see whether 'printer'  button has been selected
    cmpi.w    #1,(a0)

    move.l    #rsc_free,aespb    ; free memory taken up by the
* resource file
    bsr       aes
    bra       exit               ; let's quit

obdraw:
    move      #0,intin
    move      #2,intin+2
    move      d0,intin+4
    move      d1,intin+6
    move      d2,intin+8
    move      d3,intin+10
    move.l    parent,addrin
    move.l    #object_draw,aespb
    bsr       aes
    rts

*   .globl    f_do
f_do: move.l  #form_do,aespb      ; form_do
    move      #date,intin         ; editable text field
    move.l    parent,addrin
    bsr       aes
    rts

form_center:
    move.l    #f_center,aespb
    move.l    parent,addrin
```

```
    jsr          aes
    movem.w      intout+2,d0-d3
    rts

* AES subroutine
aes: move.l      #aespb,d1
    move.l       #$c8,d0
    trap         #2
    rts

exit:
    clr.w        -(sp)
    trap         #1

        ds.l  256
ustk:   ds.l  1

aespb:  dc.l   contrl,global,intin,intout,addrin,addrout

object_draw:     dc.w   42,6,1,1,0
form_do:         dc.w   50,1,2,1,0
f_center:        dc.w   54,0,5,1,0
rsc_load:        dc.w   110,0,1,1,0
rsc_gaddr:       dc.w   112,2,1,0,1
rsc_free:        dc.w   111,0,1,0,1

contrl:          ds.w  12
intin:           ds.w   128
intout:          ds.w   128
global:          ds.w   16
addrin:          ds.w   128
addrout:         ds.w   128

parent:          ds.l  1

rsc_file:    dc.b  "example.rsc",0
```

Along with the '.RCS' file and '.H' file a '.DEF' file is also created when the resource file is saved. Although not needed by GEM when using a resource file in a program it is needed by the RCP when further editing is necessary.

Using the names of objects

'form_do' needs the index number of the first editable object as one of its parameters. If there is no editable text object then the first word of the 'intin' array should be cleared. As there is an editable text object in the above source code it is easy to use the 'date' equate found in the '.H' file. We do not even have to know what the actual value of 'date' is.

Another way that the names of objects can be used is when we need to know what the user has actually selected after exiting the dialog box. In the dialog box above a printer or modem could have been selected, or one of the three buttons, but how are we to know?

If 'Cancel' has been selected then it is up to us to restore the dialog back to its prior state. This is the normal use of the 'Cancel' button. To do this we need a list of the condition it was in prior to the dialog box being used. This would be done by checking the object's status and storing its state prior to the dialog's use. See next example source code for a demonstration of this.

As we know each object is defined by a 24 byte structure, with the object status 10 bytes ahead of that, and fortunately these are sorted into index order by GEM when they are loaded so that using the formula:

```
move.l    parent,a0               ; put address of tree in A0
add.l     #('object'*24)+10,a0    ; object equate*24+10 bytes
cmpi.w    #1,(a0)                 ; see if selected
```

can show whether an object has been selected or not.

When the 'printer' button status is tested it takes the form of:

```
    move.l  parent,a0       ; put address of tree in A0
    add.l  #(printer*24)+10,a0    ; get address of object status- 'printer'
* test to see whether 'printer'  button has been selected
    cmpi.w  #1,(a0)
```

If the 'printer' object had been selected by the user then its status would be 'selected'. We can then decide what needs to be done. For instance selecting 'printer' may be a signal to the program that something needs to be printed.

What if we needed to have the 'printer' object selected as a default state, and for instance 'button1' selected too, as a default. We cannot set the object status flag to selected within the RCP, but we can set the flags before we display the dialog box in assembler using the knowledge we already have. GEM6.S shows how this may be done.

To summarize the process of displaying a dialog box:

1. Load the resource file from disk into memory, by 'rsrc__load' call.

2. Find address of object tree with 'rsrc__gaddr' call.

3. Call 'form__center' to get centred coordinates of dialog box.

4. Call 'form__dial' to reserve screen memory space. (optional)

5. Call 'form__dial' again to draw a growing box. (optional)

6. Draw the dialog box with 'objc__draw' using centred coordinates from (3)

7. Call 'form__do'. The AES now assumes complete control over user interaction with the dialog box until the user clicks on an 'exit' object or presses the Return key to activate a 'default' object. If 'objc__edit' is used instead of 'form__do' the programmer gas to take control of some of the functions that 'form__do' would normally handle.

8. Remove dialog box from screen by calling 'form__dial'. (optional)

9. Show a shrinking box by calling 'form__dial' again. (optional)

To display the dialog box again step 1 and 2 are not necessary.

For example code using 'form__dial' see next chapter.

GEM6.S *shows how objects can have their status flags set in a dialog* prior to being shown on screen, eg showing objects in a default selected state. It also demonstrates returning a dialog box back to its original state when the 'Cancel' button is selected. In actual practise a dialog box would be returned back to its state before it was invoked if the cancel button was selected. This may or may not be the default state. This would depend on whether the dialog box set-up was altered previously and exited with an 'Ok' button being selected. A complicated dialog box may well have a 'default' button so that it could be returned back to its original boot-up state. Some programs even offer the user the option of saving the users own defaults in a file often called something like 'DEFAULT.DEF'. At boot-up the programmer will then load this file automatically and set the dialog accordingly. For instance a dialog box might offer the choice of serial or parallel printer, flashing cursor or still cursor, etc. The user then selects his preferences and the saves them.

```
* GEM6.S
* Load and display an example Resource file: EXAMPLE.RSC
* Use AES calls only. The mouse pointer is also changed to an
* arrow.
* Equates modified from the file EXAMPLE.H
* NOTE resource file must be in root directory of drive program run
* from. Set object spec, and reset object spec if 'cancel' selected.

* equates from EXAMPLE.H
tree001         equ     0
cancel          equ     2
button2         equ     4
button1         equ     5
button3         equ     6
date            equ     9
modem           equ     11
printer         equ     12
ok              equ     14

* header
    move.l      7,a5
    move.l      #ustk,a7
    move.l      4(a5),a5
    move.l      12(a5),d0
    add.l       20(a5),d0
```

```
    add.l    28(a5),d0
    add.l    #$100,d0
    move.l   d0,-(sp)
    move.l   a5,-(sp)
    clr.w    -(sp)
    move     #$4a,-(sp)
    trap     #1
    add.l    #12,sp

* appl_intit()
    move.l   #appl_init,aespb
    jsr      aes      ; call AES

    move.l   #rsc_load,aespb    ; AES load a resource file
    move.l   #rsc_file,addrin   ; name of resource file to  be
* loaded
    jsr      aes
    cmpi.w   #0,intout          ; was the resource file loaded
    beq      exit               ; no

    move.l   #rsc_gaddr,aespb   ; get address of resource tree
    move     #0,intin           ; get whole tree structure
    move     #tree001,intin+2   ; tree
    bsr      aes
    cmpi.w   #0,intout          ; error
    beq      exit               ; yes

    move.l   addrout,parent     ; place address in parent

    move.l   parent,a0          ; address in a0
    move     #(button1*24+10),d0    ; offset in d0
    move     #1,0(a0,d0)            ; make button1 default selected
    move     0(a0,d0),but1_status   ; save status

    move     #(button2*24+10),d0
    move     0(a0,d0),but2_status   ; save status

    move     #(button3*24+10),d0
    move     0(a0,d0),but3_status   ; save status

    move     #(printer*24+10),d0
```

```
    move      #1,0(a0,d0)          ; make printer default selected
    move      0(a0,d0),printer_status  ; save status

    move      #(modem*24+10),d0
    move      0(a0,d0),modem_status   ; save status

    bsr       arrow               ; change mouse to arrow
    bsr       form_center         ; get centred coords of dialog
* box
    bsr       obdraw              ; draw it on screen
    bsr       f_do                ; handle interaction with user

    move.l    parent,a0

    add.l     #(cancel*24)+10,a0    ; get address of object status-
* 'cancel'

* test to see whether 'cancel' button has been selected
    cmpi.w    #1,(a0)
    bne       dont_restore

* restore status of buttons

    move.l    parent,a0           ; place address in a0
    move      #(button1*24+10),d0    ; offset in d0
    move      but1_status,0(a0,d0)   ; restore old status

    move      #(button2*24+10),d0
    move      but2_status,0(a0,d0)

    move      #(button3*24+10),d0
    move      but3_status,0(a0,d0)

    move      #(printer*24+10),d0
    move      printer_status,0(a0,d0)

    move      #(modem*24+10),d0
    move      modem_status,0(a0,d0)

dont_restore:
    bsr       obdraw
```

```
        bsr         f_do
        move.l      #rsc_free,aespb   ; free memory taken up by  the
* resource file
        bsr         aes
        bra         exit              ; let's quit

obdraw:
        move        #0,intin
        move        #2,intin+2
        move        cx,intin+4
        move        cy,intin+6
        move        cw,intin+8
        move        ch,intin+10
        move.l      parent,addrin
        move.l      #object_draw,aespb
        bsr         aes
        rts

*       .globl      f_do
f_do:   move.l      #form_do,aespb          ; form_do
        move        #date,intin             ; editable text field
        move.l      parent,addrin
        bsr         aes
        rts

form_center:
        move.l      #f_center,aespb
        move.l      parent,addrin
        jsr         aes
        movem.w     intout+2,d0
        movem.w     d0,cx             ; put values in cx-ch
        rts

arrow:
* graf_mouse
        movem.l     a0-a6/d0-d7,-(sp)
        move.l      #graf_mouse,aespb
        jsr         aes
        move        #0,intin          ; arrow
        movem.l     (sp)+,a0-a6/d0-d7
        rts
```

```
* AES subroutine

aes:
        movem.l     a0-a6/d0-d7,-(sp)
        move.l      #aespb,d1
        move.l      #$c8,d0
        trap        #2
        movem.l     (sp)+,a0-a6/d0-d7
        rts

exit:
* appl_exit()
        move.l      #appl_exit,aespb
        jsr         aes         ; call AES

        clr.w       -(sp)
        trap        #1


        ds.l        256
ustk:   ds.l        1

aespb:  dc.l    contrl,global,intin,intout,addrin,addrout

object_draw:    dc.w    42,6,1,1,0

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
form_do:        dc.w    50,1,2,1,0
f_center:       dc.w    54,0,5,1,0
graf_mouse:     dc.w    78,1,1,1,0
rsc_load:       dc.w    110,0,1,1,0
rsc_gaddr:      dc.w    112,2,1,0,1
rsc_free:       dc.w    111,0,1,0,1

contrl:         ds.w    12
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    16
addrin:         ds.w    128
addrout:        ds.w    128
```

```
parent:         ds.l    1
rsc_file:       dc.b    "example.rsc",0

* these 4 must stay together
cx:     ds.w    1
cy:     ds.w    1
cw:     ds.w    1
ch:     ds.w    1

but1_status:    ds.w    1
but2_status:    ds.w    1
but3_status:    ds.w    1

printer_status: ds.w    1
modem_status:   ds.w    1
```

GEM6.S is a, slightly unusual program in that 'form__do' is executed twice. Under normal program conditions we have to expect a dialog box to be called many times, and it is for this purpose that GEM6.S has been written to show what happens when objects are selected and the dialog box is exited from.

GEM helps in many ways to ease the programming burden but it expects the programmer to see to it that objects are returned to their original conditions if necessary. For instance it is usual to ensure that the 'OK' or 'Cancel' button once selected are returned to their non-selected state. GEM does not do this automatically. In the program above I have shown how it is possible to reset all the objects back to their original condition when the 'Cancel' button is selected. But you may have noticed that I have not taken care of the 'Ok' or 'Cancel' button state, so that once it has been selected it stays selected. It could have been easily turned back to its original state in the same manner as all the other selectable objects, but they have been left as an example.

GEM provides a call to alter an objects status called 'objc__change', which can be used as an alternative to the methods outlined above, although it is not as flexible.

It may be useful to examine this program fragment in more detail:

```
move.l   parent,a0              ; address in a0
move     #(button1*24+10),d0    ; offset in d0
move     #1,0(a0,d0)            ; make button1 default selected
move     0(a0,d0),but1_status   ; save status
```

First the address of the dialog box tree is placed in address register a0, then the index of 'button1' is multiplied by 24 to get the address of that particular object. To get the object status a further 10 (bytes) needs to be add to that result. This is then placed in d0. The third line takes the address in a0, adds whatever is in d0 to it, and places one in the place referred to by that address so that button1 is now selected. Fortunately, register a0's contents are not altered by this operation so that it is possible to alter the value of d0 to get further addresses. The last line stores the new contents of this address and places it at the address labelled 'but1_status' ready for use if the 'Cancel' button is selected by the user.

Note that the mouse pointer is altered to an arrow from the busy bee, with the 'graf_mouse' AES call. Other options are available to the programmer, such as a pointing hand. See disk for list of options and chapter fifteen.

Sorting objects

If a group of objects is created with the RCP, such as a group of editable objects then we would probably need the objects to be in order so that for instance pressing the up arrow key sends the cursor up to the last object, and so on. However, it is often the case that such a group of objects is not ordered correctly in the process of creating the dialog box. In fact they may be ordered in a seemingly haphazard way, which can be due to a number of factors eg the 'copy' option being used. Fortunately the RCP has a 'Sort' option which permits the ordering of a group of siblings, or children of the same object.

For instance if a group of editable objects where laid out like this:

```
First name     :____________   Last name      :____________

Home Address   :____________   Work Address   :____________
               :____________                  :____________
               :____________                  :____________
```

	:___________		:___________
Post Code	:_____ _____	Post Code	:_____ _____

As we know the Tab key or down arrow will take the cursor to the next editable object (actually the object pointed to by that object which could be the next physical field but might not). When the user types his/her name into the 'First name' field the cursor should logically go next to the 'Last name' field, but if the objects are arranged so that the second column follows on from the first column then 'Home Address' will be the next input field. Not what is wanted.

The 'Sort' option allows all the siblings of an object (at one level) to be sorted in a four different ways. 'X only', 'Y only', 'X then Y', and 'Y then X'.

'X' refers to the layout of objects in columns whilst 'Y' refers to objects in rows. So sorting all the siblings in the order 'X' only would result in the cursor following the objects in columns, 'first name' followed by 'home address' etc. Sorting by 'Y only' results in the cursor following the objects in rows:

```
---field 0------>    ---field 1------>
---field 2------>    ---field 3------>   etc
```

In this case the cursor would go to field 1 then to 2 etc.

Sorting 'X then Y' is the same as 'X only' in this case, and 'Y then X' is the same as 'Y only' in this case too.

Last editable object

If the last object in the tree is an editable one then the ST will crash when the cursor reaches there! The solution is to ensure that the 'OK' button (for example) is the last object in the tree. In the RCP double clicking over the 'OK' button before exiting and Saving the resource file will ensure that the 'OK' button is the last object in the dialog tree. This is not the only GEM bug; 'evnt_multi' is not free of bugs. Fortunately most GEM bugs can be programmed around.

RCP Edit

The following options are available for editing objects:

X Cut
C Copy
V Paste
E Erase

They are accessed by selecting the object and then pressing ALT and the required option or by selecting the option from the drop down menu itself. For instance to copy an object it should be first selected by clicking over it, and then ALT–C should be pressed which copies the object into an internal buffer. Next 'Paste' should be selected by pressing ALT–V and where ever the mouse pointer is situated the object will be copied to. If the receiving parent object is not big enough to accommodate the pasted (copied) object then the object will not be copied until the parent is enlarged or the pasting arrow is positioned more accurately so that the object will fit into the parent.

Useful RCP options:

With the mouse pointer and left button:

Control selects parent of the object. Useful for drop down menu entries as Control–Click selecting the menu title opens up the drop down menu. Control–click on menu item selects drop down menu parent box so that it can be reduced or enlarged in size to accommodate less or more menu entries– keep control key depressed whilst opening up or reducing menu box.

Left-shift copies object to buffer.

Cursor correction

Unfortunately the AES insists on placing the cursor at the end of an editable object field. This is ok as in the last example where an example date filled the editable text field, but if the field is empty having the cursor at the end of the field is unacceptable.

To see this effect and the method to correct it please have a look at the example source code below. Diagram 13:8 shows the finished dialog box in the RCP.

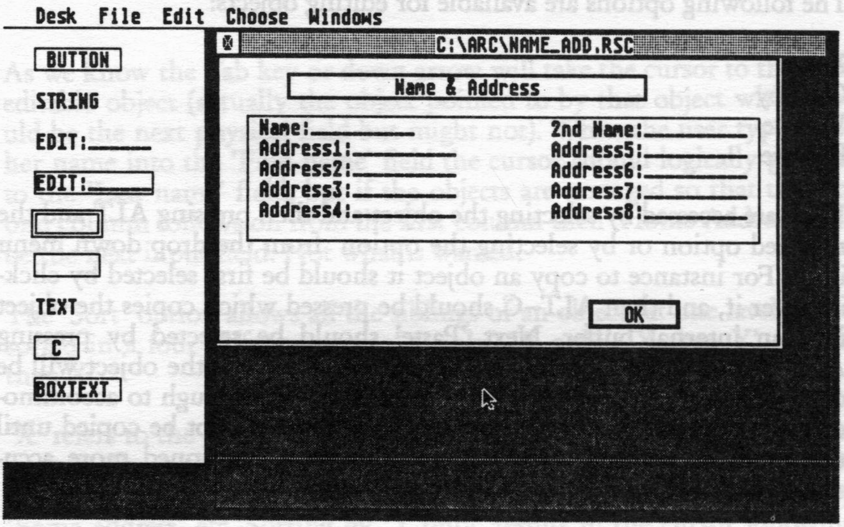

diagram 13:8

```
* GEM6A.S
* Load and display an example Resource file: NAME_ADD.RSC
* Equates modified from the file NAME_ADD.H
* NOTE: resource file must be in root directory of
* drive program run from.
* This program is used as an example of correcting cursor position
* in an editable text object field.

tree001    equ    0
name       equ    3
addr1      equ    4
addr2      equ    5
addr3      equ    6
addr4      equ    7
name2      equ    8
```

```
addr5           equ     9
addr6           equ     10
addr7           equ     11
addr8           equ     12
ok              equ     13

* header
        move.l     a7,a5
        move.l     #ustk,a7
        move.l     4(a5),a5
        move.l     12(a5),d0
        add.l      20(a5),d0
        add.l      28(a5),d0
        add.l      #$100,d0
        move.l     d0,-(sp)
        move.l     a5,-(sp)
        clr.w      -(sp)
        move       #$4a,-(sp)
        trap       #1
        add.l      #12,sp

        move.l     #rsc_load,aespb     ; AES load a resource file
        move.l     #rsc_file,addrin    ; name of resource file to  be
* loaded
        jsr        aes
        cmpi.w     #0,intout           ; was the resource file loaded
        beq        exit                ; no

        move.l     #rsc_gaddr,aespb    ; get address of resource tree
        move       #0,intin            ; tree structure
        move       #0,intin+2
        bsr        aes
        cmpi.w     #0,intout           ; error
        beq        exit                ; yes

        move.l     addrout,parent      ; place address in parent

* correct cursor
        move.l     parent,a0

* correct cursor position method 1
```

```
    move      #(name*24+12),d0

    move.l    0(a0,d0),a1

    move.l    (a1),a2
    move.b    #0,(a2)

*correct cursor position method 2

    move      #addr1,d0
    jsr       correct

    move      #addr2,d0
    jsr       correct

    move      #addr3,d0
    jsr       correct


    bsr       form_center       ; get centred coords of dialog
* box

    bsr       obdraw ; draw it on screen

    bsr       f_do              ; handle interaction with user

    move.l    parent,a0

    move.l    #rsc_free,aespb   ; free memory taken up by the
* resource file
    bsr       aes

    bra       exit              ; let's quit

correct:
    mulu      #24,d0    ; get offset
    add       #12,d0    ; get object spec
    move.l    0(a0,d0),a1   ; get address held there
    move.l    (a1),a2   ; get address pointed to by that
* address
    move.b    #0,(a2)
```

```
    rts

obdraw:
    move        #0,intin
    move        #2,intin+2
    move        d0,intin+4
    move        d1,intin+6
    move        d2,intin+8
    move        d3,intin+10
    move.l      parent,addrin
    move.l      #object_draw,aespb
    bsr         aes
    rts

*   .globl      f_do
f_do: move.l    #form_do,aespb      ; form_do
    move        #name,intin         ; editable text field
    move.l      parent,addrin
    bsr   aes
    rts

form_center:
    move.l      #f_center,aespb
    move.l      parent,addrin
    jsr         aes
    movem.w     intout+2,d0-d3
    rts

* AES subroutine
aes: move.l     #aespb,d1
    move.l      #$c8,d0
    trap        #2
    rts

exit:
    clr.w       -(sp)
    trap        #1

        ds.l  256
ustk:   ds.l  1
```

```
aespb:          dc.l  contrl,global,intin,intout,addrin,addrout
object_draw:    dc.w  42,6,1,1,0
form_do:        dc.w  50,1,2,1,0
f_center:       dc.w  54,0,5,1,0
rsc_load:       dc.w  110,0,1,1,0
rsc_gaddr:      dc.w  112,2,1,0,1
rsc_free:       dc.w  111,0,1,0,1

contrl:         ds.w  12
intin:          ds.w  128
intout:         ds.w  128
global:         ds.w  16
addrin:         ds.w  128
addrout:        ds.w  128

parent:         ds.l  1

rsc_file:       dc.b  "name_add.rsc",0
```

If you assemble and run the above example you will be able to see that only those objects that have a correction made to the tedinfo structure have the cursor in the right place, ie to the extreme left of the field. The correction to the tedinfo structure is to place a null at the fist word of 'te__ptext' ensuring that this is seen as the end of the field, Without this correction the cursor is placed at the extreme right end of the editable field.

In the above example only 'name', 'addr1', 'addr2', and 'addr3' are corrected in this manner. This bug is not the fault of MKRSC.PRG, other RCP's exhibit the same fault. The bug is a GEM one and is a nuisance but can be easily corrected with the code given above.

Examining the code closely to see how it is done we have to aware that the object spec points to a tedinfo data structure which itself refers to an address. We can see this in the code taken from chapter fifteen. It is not necessary at this point to understand all this information but it is put here for reference.

```
    dc.w    6,-1,-1,29,8,0
    dc.l    addr5           ; object spec points to addr5
    dc.w    92,45,400,15
```

```
addr5: dc.l   tx5,txtg4,txt2   ; addr5 (te_ptext) points to tx5
       dc.w   3,0,0,$11f0,0,0,37,37

tx5:   dc.b   '@                    ',0 ; actual string to be output
```

So examining the code taken from GEM6A.S to correct the cursor position:

```
* correct cursor position method 1
    move      #(name*24+12),d0
    move.l    0(a0,d0),a1
    move.l    (a1),a2
    move.b    #0,(a2)
```

The first line calculates the offset position of name, and adds 12 bytes to the position to get the object specification offset. This is then added in line two (without affecting a0) to the address in a0 which is the start address of the object tree. The resultant address is passed to a1. But a1 points to or refers to another address the string 'te__pext' outputs. So this is passed to register a2, and finally at this address a null is placed or in BASIC poked. The 'correct:' subroutine does the same thing.

In GFA BASIC the code for correcting this fault is:

```
POKE LPEEK(LPEEK(tree%+(sourcename*24)+12)),0
```

Where 'tree%' is the address of the tree loaded by 'rsc__load' and 'sourcename' holds the index of the editable object.

GFA BASIC V3 users can use a slightly different method to achieve the same result:

```
CHAR{{OB_SPEC(tree%,sourcename)}}=""
```

When creating editable text fields in the RCP the underline key to the left of the '=' key, is used to donate an editable character space. The only way to see that the correct amount of underline representations have been made is to see where the cursor is placed. This is the only way to check as the RCP dialog box which accepts the underline itself uses the underline for a blank editable character as it is an editable object itself! Studying the example 'NAME_ADD.RSC' resource file in the RCP will demonstrate this to you.

Chapter 14
Drop Down Menus

This chapter gives details on how to create drop down menus using the RCP, and how to use it in an application.

Most ST application software, apart from a few notable exceptions, use the GEM WIMP interface as a means of providing a user–friendly front–end to their programs. One of the most notable features of a WIMP environment is the use of drop down menus, which can be accessed with a mouse.

It is difficult (but not impossible) to create drop down menus in assembly language by hand, so throughout this book the RCP is used create all the drop down menus. The RCP makes the creation of drop down menus a relatively painless process. GEM7.S demonstrates how to use drop down menus in an assembly language program.

Also some new GEM calls are made:

Form_dial

'form_dial' which has four modes of operation, two of which are used in the code below. First the an area the size of the dialog box is saved to an internal buffer so that when the dialog box is finished with the screen can be restored. This is a useful function although its application is restricted as it does not clean up any part of the dialog box that has infringed on a window element. In this case though we are not using a window so it is ok. The other function of 'form_dial' is to provide an expanding and shrinking box when a dialog box is opened and closed respectively. Many programmers leave this out as they feel it is something that can be profitably dispensed with as it speeds up the display of dialog boxes and windows.

Objc_change

'objc_change' is also used to alter the status of the 'OK' or 'Cancel' button when exiting the dialog box.

Form_alert

This is a standardized GEM dialog box that can handle short messages with a maximum of three exit buttons. They do not have to be created within the RCP, and they are used extensively by the operating system for 'disks full', 'data on the disk may be damaged....' and other system messages.

menu_tnormal

This GEM AES call displays a menu title in reverse video or as normal. Its main use is in de-selecting menu titles after they have been clicked on.

Menus from the RCP

A similar process to creating dialog boxes is used when creating drop down menus. One point though is that the menus should be created along with the dialog box, so that EXAMPLE.RSC contains both the dialog box and the drop down menus.

Once again it is important to name each menu entry so that the 'EXAMPLE.H' file can later be modified for inclusion in the assembly language source code.

If you have been following this book chapter by chapter you will have created a similar dialog box to the one supplied in the EXAMPLE.RSC. Now load your dialog box into the RCP, and now drag across the menu icon, and give it a name when prompted or leave it as it is as I have done. Next double click on the menu icon and two default menu titles will be displayed.

Two menu titles are created to begin with by the RCP, and they follow the GEM convention of DESK, and FILE. The next choice of menu title is usually EDIT. DESK, and FILE should not be erased from the menu bar.

The first menu title *DESK* holds the *About Message* another GEM convention where copyright messages and credits are often displayed when this menu item is selected. The rest of the menu holds the standard six desk accessories (DA). It is not necessary to alter these menu items as any DA will substitute its own name for the one in the menu automatically. GEM will also display the correct number of loaded DA's so that accessing the *DESK* drop down menu will only display the *About Message...* if there is no loaded DA's.

Another GEM convention generally adopted is the use of three periods after a menu item that displays further information when clicked on, usually in a dialog box. For example *Load...* ,which usually displays the GEM file selector dialog box.

The *About Message...* menu item should be doubled clicked and the *About Message...* should be replaced with the title of the program. I altered this to *About Example...* .

Only some of the object flags need to be selected. *Selected* should be clicked on if not already selected. Also *checked* if a tick is needed to the extreme left of the menu item. This helps to signal to the user that the menu item is in operation currently or more usually that a choice out of a list of (menu or dialog box) items has been made.

Another choice could be to disable the menu item which has been done for *Save* in the example menu. A disabled or greyed-out menu item signifies to the user that it cannot be selected until some further action is taken. For instance in a word processor it would seem sensible to restrict the use of any Save option until a file had been loaded, or a new file created in the word processor. After all there is no point in saving an empty file.

Adding menu items

To add new items (objects) to the next menu title *FILE* should be selected. If difficulty is experienced in selected a particular drop down menu then the Control key should be held down at the same time as clicking over the menu title. Once *Quit* is displayed it should be double clicked and the length of the *Quit* string shortened. Exiting from this enables us to expand the menu to accommodate more menu items by the usual process of dragging the bottom right -hand corner of the *Quit* menu box. Another (quicker) method is to hold down the Control key and click over the last item in the menu bar, in this case *Quit* This will select the parent of the menu item in this case the object box that holds the menu item(s). Keeping the Control key pressed down the box should then be expanded.

Once this has been done the *Quit* menu item can be moved about within the newly expanded menu box, and other menu items added by dragging over ENTRY to the new menu. The new menu entries should be made the same width as the menu box so that later on in an application when they are selected the whole width of the menu item is selected not just the name and perhaps a space. This is done by expanding the menu item box to the full width of the menu box, or filling in enough spaces at the end of the menu item string. Also it is normal to leave two spaces before the menu item so that ticks can be placed there if necessary. Note that any drop down menu should not be greater than one-quarter of the screen.

To alter the menu name from 'ENTRY' to what ever you want just double click over the item and edit the string in the dialog box.

To add a menu title drag across the TITLE icon and place in an appropriate position on the menu bar.

It is essential that a name is given to each menu title and entry, except the DA's. This is done by selecting each item and naming each menu object in the usual manner. It is normal to give an appropriate name to the menu objects so that they are easily recognizable in the assembly source code. The names are linked to the object's index which is invaluable later on. The names are then listed in the '.H' file which can be easily modified for inclusion in our assembly language source code.

If GEM7.S is studied it will be seen how to display a menu bar and how the mouse pointer interacts with it.

```
* GEM7.S
* Uses drop down menu, dialog box, 'objc_change', 'form_dial' and
* 'form_alert'. Needs EXAMPLE.RSC
* Equates modified from the file EXAMPLE.H
* NOTE: resource file must be in root directory of drive program run
* from Click on About Example... to display dialog box. Quit to exit
* dialog box

tree001        equ      0
```

```
cancel          equ     2
button2         equ     4
button1         equ     5
button3         equ     6
date            equ     9
modem           equ     11
printer         equ     12
ok              equ     14

* MENU
tree002         equ     1
desk            equ     3
file            equ     4
page            equ     5
about           equ     8
save_as         equ     18
load            equ     20
quit            equ     22
g_top           equ     24
g_bottom        equ     25
page0           equ     27
etc             equ     28

* a3 is used to store address for AES calls using addrin
* a4 for editable text fields if any

* header
        move.l  a7,a5
        move.l  #ustk,a7
        move.l  4(a5),a5
        move.l  12(a5),d0
        add.l   20(a5),d0
        add.l   28(a5),d0
        add.l   #$100,d0
        move.l  d0,-(sp)
        move.l  a5,-(sp)
        move    d0,-(sp)
        move    #$4a,-(sp)
        trap    #1
        add.l   #12,sp
```

```
* appl_init()
        move.l      #appl_init,aespb
        jsr         aes         ; call AES

        move.l      #rsc_load,aespb    ; AES load a resource file
        move.l      #rsc_file,addrin   ; name of resource file to be
* loaded
        jsr         aes
        cmpi.w      #0,intout          ; was the resource file loaded
        beq         exit2              ; no

* dialog box
        move.l      #rsc_gaddr,aespb   ; get address of resource tree
        move        #0,intin           ; tree structure
        move        #tree001,intin+2   ; dialog box
        bsr         aes
        cmpi.w      #0,intout          ; error
        beq         exit               ; yes

        move.l      addrout,dialog     ; place address in dialog

* menu
        move.l      #rsc_gaddr,aespb   ; get address of resource tree
        move        #0,intin           ; tree structure
        move        #tree002,intin+2   ; drop down menu menu
        bsr         aes
        cmpi.w      #0,intout          ; error
        beq         exit               ; yes

        move.l      addrout,men_bar    ; place address in men_bar

* put menu bar on screen
        move.l      #menu_bar,aespb    ; display menu object tree
        move.l      men_bar,addrin
        move        #1,intin           ; show menu_bar
        bsr         aes

        bsr         arrow              ; change mouse pointer to arrow

* evnt_mesag
evnt_mess:
```

```
    jsr        menu_t              ; change menu title to normal video
    move.l     #evnt_mesag,aespb       ; wait for report in message buffer
    move.l     #message_buffer,addrin
    bsr        aes             ; do it

* what have we got
    cmp        #10,message_buffer      ; is it a menu message? (10)
    bne        evnt_mess           ; no don't bother with it
    cmp        #desk,message_buffer+6  ; is it Desk menu bar?
    beq        do_menu             ; yes
    cmp        #file,message_buffer+6  ; is it File menu bar?
    beq        do_menu             ; yes
    jsr        do_alert            ; it's not Desk or File
    bra        evnt_mess

do_menu:
    cmp        #about,message_buffer+8  ; has About... been
* selected?
    beq        got_about           ; yes
    cmp        #quit,message_buffer+8   ; has Quit been selected?
    beq        exit
    jsr        do_alert            ; neither selected
    bra        evnt_mess

got_about:
    move       #0,form_flag    ; reserve area of screen memory
    move.l     dialog,a3       ; address of dialog tree in a3
    move       #date,d4        ; date object in d4

    jsr        do_dialog       ; display dialog box and interact with it

    move       #3,form_flag    ; release area of screen memory
    jsr        form_d          ; do it
    bra        evnt_mess

do_dialog:
    bsr        form_center         ; get centred coords of dialog
* box

    bsr        form_d          ; reserve screen memory
    bsr        obdraw          ; draw it on screen
```

```
	bsr	f_do		; handle interaction with user
	bsr	ob_change	; reset ok or cancel to non selected

	rts

****** subroutines *******
do_alert:
	move.l	#form_alert,aespb
	move	#1,intin		; first button
	move.l	#alert_string,addrin
	bsr	aes
	rts

ob_change:
	move.l	#objc_change,aespb
	move	intout,intin	; ok or cancel- from 'form_do'
	move	#0,intin+2
	move	cx,intin+4
	move	cy,intin+6
	move	cw,intin+8
	move	ch,intin+10
	move	#0,intin+12	; new status- not selected
	move	#1,intin+14	; not re-drawn after status change
	move.l	a3,addrin
	bsr	aes
	rts

menu_t:
	move.l	#menu_tnormal,aespb
	move	message_buffer+6,intin
	move	#1,intin+2
	move.l	men_bar,addrin
	bsr	aes
	rts

obdraw:
	move	#0,intin
	move	#2,intin+2
	move	cx,intin+4
	move	cy,intin+6
```

```
        move        cw,intin+8
        move        ch,intin+10
        move.l      a3,addrin
        move.l      #object_draw,aespb
        bsr         aes
        rts

*       .globl      f_do
* a4 contains editable text field if any
f_do:   move.l      #form_do,aespb      ; form_do
        move        d4,intin            ; editable text field
        move.l      a3,addrin
        bsr         aes
        rts

* form_dial
form_d:
        move        form_flag,intin
        move        cx,intin+2
        move        cy,intin+4
        move        cw,intin+6
        move        ch,intin+8
        move        cx,intin+10
        move        cy,intin+12
        move        cw,intin+14
        move        ch,intin+16
        move.l      #form_dial,aespb
        bsr         aes
        rts

form_center:
        move.l      #f_center,aespb
        move.l      a3,addrin
        bsr         aes
        movem.w     intout+2,d0-d3
        movem.w     d0-d3,cx
        rts

* AES subroutine
aes:    move.l      #aespb,d1
        move.l      #$c8,d0
```

```
    trap        #2
    rts

arrow:
* graf_mouse
    movem.l     a0-a6/d0-d7,-(sp)
    move.l      #graf_mouse,aespb
    move        #0,intin            ; arrow
    bsr         aes
    movem.l     (sp)+,a0-a6/d0-d7
    rts

exit:
    move.l      #rsc_free,aespb     ; release memory taken up by the
* resource file
    bsr         aes

exit2:

* appl_exit()
    move.l      #appl_exit,aespb
    bsr         aes         ; call AES

    clr.w       -(sp)
    trap        #1
    addq.l      #2,sp


      ds.l   256
ustk: ds.l   1

aespb: dc.l  contrl,global,intin,intout,addrin,addrout

object_draw:     dc.w   42,6,1,1,0
form_do:         dc.w   50,1,2,1,0
f_center:        dc.w   54,0,5,1,0
menu_bar:        dc.w   30,1,1,1,0
evnt_mesag:      dc.w   23,0,1,1,0
form_dial:       dc.w   51,9,1,1,0
```

```
appl_init:        dc.w  10,0,1,0,0
appl_exit:        dc.w  19,0,1,0,0

rsc_load:         dc.w  110,0,1,1,0
rsc_gaddr:        dc.w  112,2,1,0,1
rsc_free:         dc.w  111,0,1,0,1

graf_mouse:       dc.w  78,1,1,1,0
menu_tnormal:     dc.w  33,2,1,1,0
objc_change:      dc.w  47,8,1,1,0
form_alert:       dc.w  52,1,1,1,0

message_buffer:   ds.b  16

* these 4 must stay together
cx:     ds.w      1
cy:     ds.w      1
cw:     ds.w      1
ch:     ds.w      1

contrl:           ds.w  128
intin:            ds.w  128
intout:           ds.w  128
global:           ds.w  128
addrin:           ds.w  128
addrout:          ds.w  128

dialog:           ds.l  1
men_bar:          ds.l  1
form_flag:        ds.w  1

rsc_file:         dc.b  "example.rsc",0

alert_string:     dc.b  "[3][There is nothing | assigned to this |"
                  dc.b " menu entry! | Please try another.][ Why not? ]",0
```

To use 'form__dial' to show an expanding and shrinking box it could be used as described below. Note that it should be used prior to 'objc__draw' and after exiting 'form__do' respectively.

```
    move        #1,flag       ; expanding box
```

```
    bsr         form_di

*** rest of routine *****

    move        #2,flag        ; shrinking box
    bsr         form_di


* form_dial- expanding/shrinking box from centre of screen.
form_di:
    move        flag,intin         ; expanding (1)/shrinking box (2)
    move        #319,intin+2        ; ok for med and hi res; x
 * coord for rectangle in its smallest size
    move        #199,intin+4        ; should be halved for med res;
* ditto y coord
    move        #0,intin+6         ; ditto width
    move        #0,intin+8         ; ditto height
    move        cx,intin+10        ; x coord of rectangle in its  largest size
    move        cy,intin+12        ; ditto y
    move        cw,intin+14        ; ditto width
    move        ch,intin+16        ; ditto height
    move.l      #form_dial,aespb
    bsr         aes
    rts

flag            ds.w 1
```

It is also possible to get the expanding box to start from a particular menu item, and shrink back to it, by using the menu coordinates.

'evnt__mesag' is one of the calls that make up the 'evnt__multi' call. 'evnt__mesag' waits until a report is present in the event buffer. There are many message events most of which concern GEM windows. For instance a message might be that the user has clicked the full box in order to enlarge the window to its full size, or reduce it to its former size. However, 'evnt__mesag' returns through 'intout' the report that the user has selected an option from one of the available drop down menus (mn__selected):

mn__selected:

16 byte buffer passed to 'evnt_mesag'

word 0= 10 if a drop down menu has been clicked on
word 3= object index of the menu title
word 4= object index of the menu entry

From this it is easy to check what menu entry has been selected especially as the equates at the start of the program allow us to use the name of each menu item as we check which one it is.

Please see disk and chapter fifteen for more coverage of message events.

```
    move      #319,intin+2     ; ok for med and hi res, x
* coord for rectangle in its smallest size
    move      #199,intin+4     ; should be halved for med res
* ditto y coord
    move      #0,intin+6       ; ditto width
    move      #0,intin+8       ; ditto height
    move      cx,intin+10      ; x coord of rectangle in its largest size
    move      cy,intin+12      ; ditto y
    move      cw,intin+14      ; ditto width
    move      ch,intin+16      ; ditto height
    move.l    #form_dial,aespb
    bsr       aes
    rts

flag          dc.w 1
```

It is also possible to get the expanding box to start from a particular menu item, and shrink back to it, by using the menu coordinates.

'evnt_mesg' is one of the calls that make up the 'evnt_multi' call. 'evnt_mesg' waits until a report is present in the event buffer. There are many message events most of which concern GEM windows. For instance a message might be that the user has clicked the full box in order to enlarge the window to its full size, or reduce it to its former size. However, 'evnt_mesg' returns through 'intout' the report that the user has selected an option from one of the available drop down menus (mn_selected):

mn_selected:

Chapter 15
Editing Text

This chapter looks at creating a GEM dialog box by hand in which text can be freely edited. The created dialog box which asks the user for name and addresses and other particulars to be input is something that might be seen in an accounts or database program. The use of 'evnt__multi' and 'objc__edit' is also looked at. Using 'evnt__multi' with drop down menus is also examined.

The following program (for high and medium res) demonstrates the use of a hand made dialog box which allows characters to be freely edited.

```
* GEM8.S
* This program displays a dialog box into which text can be freely
* entered and edited. The mouse can be used to position the cursor.
* Pressing the Return key ends all editing.

* header
    move.l      a7,a5
    move.l      #ustk,a7
    move.l      4(a5),a5
    move.l      12(a5),d0
    add.l       20(a5),d0
    add.l       28(a5),d0
    add.l       #$100,d0
    move.l      d0,-(sp)
    move.l      a5,-(sp)
    clr         -(sp)
    move        #$4a,-(sp)
    trap        #1
    add.l        #12,sp

* appl_init()
    move.l      #appl_init,aespb
    jsr         aes         ; call AES

* get current screen res
```

```
        move        #4,-(sp)
        trap        #14
        addq.l      #2,sp
* res returned in d0

        cmp         #2,d0       ; is it high res
        bne         dont_alter_coords

        move.l      #parent,a5
        move.l      #11,d5
        bsr         alter_coords

dont_alter_coords:
        bsr         f_center

        bsr         obdraw
        bsr         f_do
        bra         exit

alter_coords:
        add.l       #18,a5
        move.w      (a5),d3
        mulu.w      #2,d3
        move        d3,(a5)+
        add.l       #2,a5
        move        (a5),d3
        mulu.w      #2,d3
        move        d3,(a5)+
        dbf         d5,alter_coords
        rts

obdraw:
        move        #0,intin
        move        #1,intin+2
        move        cx,intin+4
        move        cy,intin+6
        move        cw,intin+8
        move        ch,intin+10
        move.l      #parent,addrin
        move.l      #objc_draw,aespb
```

```
        bsr         aes
        rts

f_do:   move.l      #form_do,aespb
        move.w      #1,intin          ; first editable object
        move.l      #parent,addrin
        bsr         aes
        rts

f_center:
        move.l      #form_center,aespb
        move.l      #parent,addrin
        jsr         aes
        move.w      intout+2,cx
        move.w      intout+4,cy
        move.w      intout+6,cw
        move.w      intout+8,ch
        rts

aes:    move.l      #aespb,d1
        move.l      #$c8,d0
        trap        #2
        rts

exit:
* appl_exit()
        move.l      #appl_exit,aespb
        bsr         aes               ; call AES

        clr.w       -(sp)
        trap        #1


        ds.l  256
ustk:   ds.l  1

addr1:  dc.l  tx1,txtg1,txt2
        dc.w  3,0,0,$11f0,0,0,37,37

tx1:    dc.b  '@                        ',0
```

```
* need @ so that cursor is to left

txtg1:  dc.b   'Code     :______________________________________',0

txt2:   dc.b   'nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn',0

addr2:  dc.l   tx2,txtg2,txt2
        dc.w   3,0,0,$11f0,0,0,37,37

tx2:    dc.b   '@                              ',0

txtg2:  dc.b   'Name     :______________________________________',0


addr3:  dc.l   tx3,txtg3,txt2
        dc.w   3,0,0,$11f0,0,0,37,37

tx3:    dc.b   '@                              ',0

txtg3:  dc.b   'Address  :______________________________________',0


addr4:  dc.l   tx4,txtg4,txt2
        dc.w   3,0,0,$11f0,0,0,37,37

tx4:    dc.b   '@                              ',0

txtg4:  dc.b   ':_____________________________________',0

addr5:  dc.l   tx5,txtg4,txt2
        dc.w   3,0,0,$11f0,0,0,37,37

tx5:    dc.b   '@                              ',0

addr6:  dc.l   tx6,txtg4,txt2
        dc.w   3,0,0,$11f0,0,0,37,37

tx6:    dc.b   '@                              ',0

addr7:  dc.l   tx7,txtg4,txt2
        dc.w   3,0,0,$11f0,0,0,37,37
```

```
tx7:    dc.b   '@                    ',0

addr8:  dc.l   tx8,txtg8,txt2
        dc.w   3,0,0,$11f0,0,0,37,37

tx8:    dc.b   '@                    ',0

txtg8:  dc.b   'Post Code:_______________________________________',0

addr9:  dc.l   tx9,txtg9,txt2
        dc.w   3,0,0,$11f0,0,0,37,37

tx9:    dc.b   '@                    ',0

txtg9:  dc.b   'Phone No :_______________________________________',0

addr10: dc.l   tx10,txtg10,txt2
        dc.w   3,0,0,$11f0,0,0,37,37

tx10:   dc.b   '@                    ',0

txtg10: dc.b   'VAT No   :_______________________________________',0

texttq:         dc.b    0

ok_text:        dc.l    text_ok,texttq,texttq
                dc.w    3,0,2,$11f0,0,1,7,0

text_ok:        dc.b    ' OK ',0

parent: dc.w    -1,1,11,20,0,16
        dc.l    $00021100
        dc.w    50,30,450,125

        dc.w    2,-1,-1,29,8,0
        dc.l    addr1
        dc.w    20,5,400,15

        dc.w    3,-1,-1,29,8,0
```

```
        dc.l    addr2
        dc.w    20,15,400,15

        dc.w    4,-1,-1,29,8,0
        dc.l    addr3
        dc.w    20,25,400,15

        dc.w    5,-1,-1,29,8,0
        dc.l    addr4
        dc.w    92,35,400,15

        dc.w    6,-1,-1,29,8,0
        dc.l    addr5
        dc.w    92,45,400,15

        dc.w    7,-1,-1,29,8,0
        dc.l    addr6
        dc.w    92,55,400,15

        dc.w    8,-1,-1,29,8,0
        dc.l    addr7
        dc.w    92,65,400,15

        dc.w    9,-1,-1,29,8,0
        dc.l    addr8
        dc.w    20,75,400,15

        dc.w    10,-1,-1,29,8,0
        dc.l    addr9
        dc.w    20,85,400,15

        dc.w    11,-1,-1,29,8,0
        dc.l    addr10
        dc.w    20,95,400,15

        dc.w    0,-1,-1,22,7+32,0
        dc.l    ok_text         ;       OK
        dc.w    150,110,60,10

aespb:  dc.l    contrl,global,intin,intout,addrin,addrout
```

```
objc_draw:      dc.w    42,6,1,1,0
form_do:        dc.w    50,1,2,1,0
form_center:    dc.w    54,0,5,1,0

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0

contrl:         ds.w    12
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    16
addrin:         ds.w    128
addrout:        ds.w    128

* these 4 need to stay together
cx:     ds.w    1
cy:     ds.w    1
cw:     ds.w    1
ch:     ds.w    1
```

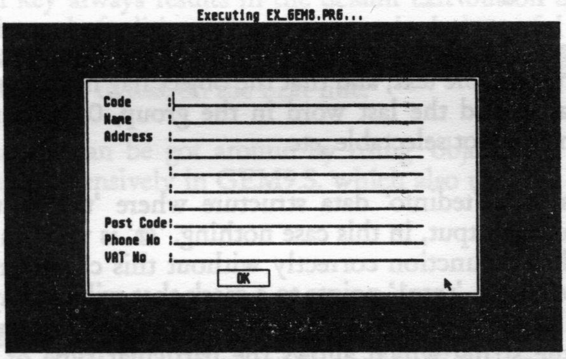

diagram 15:1 The dialog box

To understand this program we need to look carefully at the object data, and 'tedinfo' data structures

```
* first child of dialog box (object data)
```

```
        dc.w     2,-1,-1,29,8,0 ; g_ftext, editable, normal
        dc.l     addr1 ; this points to tedinfo data
        dc.w     20,5,400,15    ; x,y width & height

* tedinfo data

addr1:  dc.l     tx1,txtg1,txt2
        dc.w     3,0,0,$11f0,0,0,37,37

tx1:    dc.b  '@                                    ',0

txtgl:  dc.b  'Code    :____________________________________',0

txt2:   dc.b  'nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn',0
```

The object data's first three words state that the next object (index) is two, and there are no children (no subordinate starting object= −1, no subordinate ending object= −1)

The next three (29,8,0) words state that the object is a type 'g__ftext' which supports editable text, and that the object flag is set to so that the object is editable, and the last word in the group, 0, gives the object status as normal, ie not selectable ,etc.

'addr1' points to a 'tedinfo' data structure where 'tx1' is the actual editable text to be output, in this case nothing. '@' is very important as the object will not function correctly without this character placed at the start of the string. 'txtg1' points to a mask that will overlay the 'tx1' string and will therefore used as a template for the input of text. 'txt2' is a pointer to the string which allows the particular type of characters that can be entered into the dialog box.

But what about this bit of code?

```
alter_coords:
* a5 holds address of tree
* d5 number of of objects in tree
      add.l     #18,a5 ; add 18 (bytes) to bring to y coordinate
      move.w    (a5),d3 ; put contents of that address in d3
```

```
        mulu.w      #2,d3   ; double it to satisfy higher resolution
        move        d3,(a5)+ ; put back, and increment
        add.l       #2,a5   ; increment again
        move        (a5),d3 ; repeat process for height
        mulu.w      #2,d3
        move        d3,(a5)+
        dbf         d5,alter_coords    ; not end of tree then do again
        rts
```

The coordinates in the object data, specifically the object 'y' and 'height' coordinates are suitable only for medium resolution, so they must be altered for high resolution by multiplying them by two. This is because med res screen height as measured in the usual way (pixels) has a height of 200, whilst high res has a height of 400. This code ensures that this happens and that the dialog box is suitable for high res screen output. The dialog box was originally constructed in med res. The med res screen width is the same as high res 0–639 pixels.

One of the drawbacks of this type of GEM dialog box is that pressing the Return key always results in the default exit button being selected, and thus the end of editing. This is not particularly useful as most users would expect a 'Return' to signal that the cursor would go to the next line, or to the end of text on the next line.

This limitation can be got around by using 'objc__edit' an AES call, which is used extensively in GEM9.S, which also uses the 'evnt__multi' AES call:

```
* GEM9.S
* This program displays a dialog box into which text can be freely
* entered and edited. The mouse can be used to click over the
* OK button to exit. This program demonstrates the use of
* 'evnt_multi', and 'objc_edit'
* Pressing the HELP key ends all editing.

* header
        move.l      a7,a5
        move.l      #ustk,a7
        move.l      4(a5),a5
        move.l      12(a5),d0
```

```
        add.l       20(a5),d0
        add.l       28(a5),d0
        add.l       #$100,d0
        move.l      d0,-(sp)
        move.l      a5,-(sp)
        clr         -(sp)
        move        #$4a,-(sp)
        trap        #1
        add.l       #12,sp

* appl_init()
        move.l      #appl_init,aespb
        jsr         aes         ; call AES

* get current screen res
        move        #4,-(sp)
        trap        #14
        addq.l      #2,sp
* res returned in d0

        cmp         #2,d0               ; is it high res
        bne         dont_alter_coords   ; no

        move.l      #parent,a5          ; address of tree in a5
        move.l      #11,d5              ; number of objects
        bsr         alter_coords

dont_alter_coords:

        bsr         f_center
        bsr         obdraw
        bsr         do_text
        bra         exit

alter_coords:
* adjust object data for high res screen
        add.l       #18,a5
        move.w      (a5),d3
        mulu.w      #2,d3
        move        d3,(a5)+
        add.l       #2,a5
```

```
        move        (a5),d3
        mulu.w      #2,d3
        move        d3,(a5)+
        dbf         d5,alter_coords
        rts

obdraw:
        move        #0,intin
        move        #1,intin+2
        move        cx,intin+4
        move        cy,intin+6
        move        cw,intin+8
        move        ch,intin+10
        move.l      #parent,addrin
        move.l      #objc_draw,aespb
        bsr         aes
        rts

f_center:
        move.l      #form_center,aespb
        move.l      #parent,addrin
        jsr         aes
        move.w      intout+2,cx
        move.w      intout+4,cy
        move.w      intout+6,cw
        move.w      intout+8,ch
        rts

* start of main text editing loop
do_text:
        move        #0,char_pos
        move        #0,key
        move        #1,index
        move.w      index,intin
        move.w      key,intin+2
        move.w      char_pos,intin+4
        move.w      #1,intin+6          ; curs on
        move.l      #parent,addrin
        move.l      #objc_edit,aespb
        jsr         aes
        move        intout+2,char_pos
```

```
        move      #0,mouse_shape          ; arrow
        jsr       graf_m
        move      #1,mouse_rect           ; leave rect(1)

evnt_mult:
        move      #7,intin       ; keyboard, mouse, #1 rectangle
        move      #1,intin+2     ; number of clicks
        move      #1,intin+4     ; left button
        move      #1,intin+6     ; left button down
        move      mouse_rect,intin+8     ; mouse enter(0)/leave(1) rectangle
        move      cx,intin+10    ; x coord #1 rectangle
        move      cy,intin+12    ; y coord #1 rect
        move      cw,intin+14    ; width
        move      ch,intin+16    ; height
        move      #0,intin+18    ;
        move      #0,intin+20    ; no coords for sec
        move      #0,intin+22
        move      #0,intin+24
        move      #0,intin+26
        move      #1,intin+28    ; timer low word
        move      #0,intin+30    ; timer high word
        move.l    #evnt_multi,aespb
        jsr       aes
        move.w    intout+2,mx             ; mouse x coord
        move.w    intout+4,my
        cmpi.w    #2,intout               ; mouse click event
        beq       mouse
        cmp       #4,intout                   ; mouse rectangle event
        beq       mouse_rectang
* must be keyboard now
        move.w    intout+10,key
        cmpi.w    #$4800,key              ; cursor  up
        beq       cursor_up

        cmpi.w    #$1c0d,key              ; Carriage Return
        beq       cr
        cmpi.w    #$5000,key              ; cursor down
        beq       cr
        cmpi.w    #$6200,key              ; HELP
        beq       help
```

```
    move.w     index,intin
    move.w     key,intin+2
    move.w     char_pos,intin+4
    move.w     #2,intin+6              ; do text
    move.l     #parent,addrin
    move.l     #objc_edit,aespb
    jsr        aes
    move.w     intout+2,char_pos
    bra        evnt_mult

help:
    rts

mouse:
*objc_find
    move       #0,intin      ; index of first obj in tree to be searched
    move       #1,intin+2    ; depth
    move       mx,intin+4    ; x coord of object
    move       my,intin+6    ; y coord of object
    move.l     #objc_find,aespb
    move.l     #parent,addrin
    jsr        aes
    cmp        #11,intout    ; OK button
    beq        do_alert
    bra        evnt_mult

do_alert:
    move.l     #form_alert,aespb
    move       #1,intin               ; first button
    move.l     #alert_string,addrin
    bsr        aes
    cmpi.w     #1,intout
    beq        evnt_mult
    rts                      ; quit

mouse_rectang:
    cmp        #1,intin+8
    beq        enter_rect
    move       #0,mouse_shape  ;arrow
    jsr        graf_m
    move       #1,mouse_rect
```

```
        bra         evnt_mult

enter_rect:
        move        #0,mouse_rect
        move        #4,mouse_shape  ; open hand
        jsr         graf_m
        bra         evnt_mult

graf_m:
        movem.l     a0-a6/d0-d7,-(sp)
        move.l      #graf_mouse,aespb
        move        mouse_shape,intin           ; arrow
        bsr         aes
        movem.l     (sp)+,a0-a6/d0-d7
        rts

cursor_up:
        cmp         #1,index        ; don't go past first line
        beq         evnt_mult
        move.b      #1,curs_up_flag

cr:
*Carriage return
        move.w      index,intin
        move.w      key,intin+2
        move.w      char_pos,intin+4
        move.w      #3,intin+6              ; cursor off
        move.l      #objc_edit,aespb
        move.l      #parent,addrin
        jsr         aes
        move        intout+2,char_pos
        move        intout+2,d0

        cmpi.b      #1,curs_up_flag         ; cursor up?
        bne         not_curs_up             ; no
        sub         #1,index                ; yes, go back one index
        bra         dont_add
not_curs_up:
        cmpi.w      #10,index             ; last editable object
        beq         dont_add
```

```
    add.w     #1,index           ; do next object
dont_add:
    move.w    index,intin
    move.w    key,intin+2
    move.w    char_pos,intin+4
    move.w    #1,intin+6         ; cursor on
    jsr       aes
    move      intout+2,char_pos
    clr.b     curs_up_flag
    bra       evnt_mult          ; and wait for another key press

aes: move.l   #aespb,d1
    move.l    #$c8,d0
    trap      #2
    rts
exit:
* appl_exit()
    move.l    #appl_exit,aespb
    bsr       aes    ; call AES

    clr.w     -(sp)
    trap      #1

        ds.l  256
ustk:   ds.l  1


addr1:  dc.l  tx1,txtg1,txt2
        dc.w  3,0,0,$11f0,0,0,37,37

tx1:    dc.b  '@                                ',0

* need @ so that cursor is to left

txtg1:  dc.b  'Code     :_________________________________',0

txt2:   dc.b  'nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn',0

addr2:  dc.l  tx2,txtg2,txt2
        dc.w  3,0,0,$11f0,0,0,37,37
```

```
tx2:    dc.b    '@                              ',0

txtg2:  dc.b    'Name    :________________________________',0

addr3:  dc.l    tx3,txtg3,txt2
        dc.w    3,0,0,$11f0,0,0,37,37

tx3:    dc.b    '@                              ',0

txtg3:  dc.b    'Address :________________________________',0

addr4:  dc.l    tx4,txtg4,txt2
        dc.w    3,0,0,$11f0,0,0,37,37

tx4:    dc.b    '@                              ',0

txtg4:  dc.b    ':________________________________',0

addr5:  dc.l    tx5,txtg4,txt2
        dc.w    3,0,0,$11f0,0,0,37,37

tx5:    dc.b    '@                              ',0

addr6:  dc.l    tx6,txtg4,txt2
        dc.w    3,0,0,$11f0,0,0,37,37

tx6:    dc.b    '@                              ',0

addr7:  dc.l    tx7,txtg4,txt2
        dc.w    3,0,0,$11f0,0,0,37,37

tx7:    dc.b    '@                              ',0

addr8:  dc.l    tx8,txtg8,txt2
        dc.w    3,0,0,$11f0,0,0,37,37

tx8:    dc.b    '@                              ',0

txtg8:  dc.b    'Post Code:_______________________________',0
```

```
addr9:  dc.l    tx9,txtg9,txt2
        dc.w    3,0,0,$11f0,0,0,37,37

tx9:    dc.b    '@                    ',0

txtg9:  dc.b    'Phone No :_____________________________________',0

addr10: dc.l    tx10,txtg10,txt2
        dc.w    3,0,0,$11f0,0,0,37,37

tx10:   dc.b    '@                    ',0

txtg10: dc.b    'VAT No    _____________________________________',0

texttq:         dc.b    0

ok_text:        dc.l    text_ok,texttq,texttq
                dc.w    3,0,2,$11f0,0,1,7,0

text_ok:        dc.b    ' OK ',3,3,0

parent: dc.w    -1,1,11,20,0,16
        dc.l    $00021100
        dc.w    50,30,450,125

        dc.w    2,-1,-1,29,8,0
        dc.l    addr1
        dc.w    20,5,400,15

        dc.w    3,-1,-1,29,8,0
        dc.l    addr2
        dc.w    20,15,400,15

        dc.w    4,-1,-1,29,8,0
        dc.l    addr3
        dc.w    20,25,400,15

        dc.w    5,-1,-1,29,8,0
        dc.l    addr4
        dc.w    92,35,400,15
```

```
        dc.w    6,-1,-1,29,8,0
        dc.l    addr5
        dc.w    92,45,400,15

        dc.w    7,-1,-1,29,8,0
        dc.l    addr6
        dc.w    92,55,400,15

        dc.w    8,-1,-1,29,8,0
        dc.l    addr7
        dc.w    92,65,400,15

        dc.w    9,-1,-1,29,8,0
        dc.l    addr8
        dc.w    20,75,400,15

        dc.w    10,-1,-1,29,8,0
        dc.l    addr9
        dc.w    20,85,400,15

        dc.w    11,-1,-1,29,8,0
        dc.l    addr10
        dc.w    20,95,400,15

        dc.w    0,-1,-1,22,7+32,0
        dc.l    ok_text         ;       OK
        dc.w    150,110,60,10

aespb:  dc.l    contrl,global,intin,intout,addrin,addrout

objc_draw:      dc.w    42,6,1,1,0
form_do:        dc.w    50,1,2,1,0
form_center:    dc.w    54,0,5,1,0

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0

objc_edit:      dc.w    46,4,2,1,0
objc_find:      dc.w    43,4,1,1,0
evnt_multi:     dc.w    25,16,7,1,0
graf_mouse:     dc.w    78,1,1,1,0
```

```
form_alert:     dc.w    52,1,1,1,0
alert_string:   dc.b    "[1][ Are you sure you want to|"
                dc.b    " leave !? ",28,29
                dc.b    "|          ",30,31,191, "|You should Save your|"
                dc.b    " work before exiting.][ No! | Exit ]",0
        even
        .bss
contrl:         ds.w    12
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    16
addrin:         ds.w    128
addrout:        ds.w    128

* these 4 need to stay together
cx:     ds.w    1
cy:     ds.w    1
cw:     ds.w    1
ch:     ds.w    1

index:  ds.w    1
key:    ds.w    1
char_pos:       ds.w    1
mx:     ds.w    1
my:     ds.w    1
mouse_rect:     ds.w    1
mouse_shape:    ds.w    1
curs_up_flag:   ds.b    1
```

GEM9.S is basically the same program as the previous one except 'form_do' has been substituted by 'objc_edit' in its various guises to get around the Return key problem. 'objc_edit' offers full and complete control over the keyboard, but the editing facilities still remain. However, the mouse has to be monitored by the user which means extra code, but overall the results are beneficial. Moving the mouse into or out of the dialog box rectangle changes the shape of the mouse pointer.

Evnt_multi

The AES call 'evnt_multi' is an all-purpose event handling routine. It can handle a variety of different events: mouse click, keyboard, mouse entry/leaving a choice of one or two rectangles, timer, and it also returns the x and y coordinates of the mouse pointer when clicked or moved in or out of a rectangle. Event messages can also be returned so that menus and windows can be monitored. A very useful call, but not without its bugs.

'evnt_multi' in the above program has been set up for keyboard, mouse click, and in or out of a rectangle.

The parameter code for these events which is passed to the first word of 'intin' is taken from this list:

bit	bit value	name	event
0	1	mu_keybd	keyboard
1	2	mu_button	mouse button or click
2	4	mu_m1	mouse rect #1
3	8	mu_m2	mouse rect #2
4	16	mu_mesag	report
5	32	mu_timer	timer

Therefore placing 7 in the first word of the intin array selects the keyboard, mouse button, and first mouse rectangle.

```
evnt_mult:
    move    #7,intin        ; keyboard, mouse, #1 rectangle
    move    #1,intin+2      ; number of clicks-ONE
    move    #1,intin+4      ; left button
    move    #1,intin+6      ; left button down
    move    mouse_rect,intin+8      ; mouse enter(0)/leave(1) rectangle
    move    cx,intin+10     ; x coord #1 rectangle
    move    cy,intin+12     ; y coord #1 rect
    move    cw,intin+14     ; width
    move    ch,intin+16     ; height
    move    #0,intin+18     ;
    move    #0,intin+20     ; no coords for sec
    move    #0,intin+22
    move    #0,intin+24
    move    #0,intin+26
    move    #1,intin+28     ; timer low word
    move    #0,intin+30     ; timer high word
```

```
move.l    #evnt_multi,aespb
jsr       aes
```

The results which are given in the first word of the intout array follow the same format as the bit arrangement passed to the 'intin' array. So checking for a mouse click event or mouse rectangle event is done simply like this:

```
cmpi.w    #2,intout          ; mouse click event
beq       mouse
cmp       #4,intout          ; mouse rectangle event
beq       mouse_rectang
```

As these two events have been checked for any other event that occurs must be a keyboard event, ie a key must have been pressed.

'evnt__multi' allows the monitoring of two rectangles so that it will recognize whether the mouse pointer has entered or left a particular rectangle. The dimensions of each rectangle are passed to the intin array. Whenever the mouse pointer enters or leaves a rectangle the evnt__multi call is invoked and the mouse position is given via 'intout+2', and 'intout+4':

```
move.w    ntout+2,mx         ; mouse x coord
move.w    ntout+4,my         ; mouse y coord
```

As 'evnt__multi' can only check whether the mouse pointer is leaving or entering a rectangle a symbol 'mouse__rect' is used to pass the correct value to intin:

```
move   mouse_rect,intin+8     ; mouse enter(0)/leave(1)
```

As soon as the mouse passes from into or out of the rectangle – in this case the dimensions of the dialog box – it passes to another routine which checks to see what value is in 'intin+8'. From this value it is decided what value needs to be passed to 'mouse__rect'– if it is 1 then pass zero, if it is zero then pass 1 to 'intin+8' and at the same time alter the shape of the mouse pointer as it crosses the boundary. Various mouse pointer shape values can be passed to 'graf__mouse' – the values used here alter the shape to an arrow or an open hand. Other types include bee (2), hand with index finger (3), plus others see disk. It is also possible for the programmer to design his/her own shape (255), and pass this to the 'graf_mouse' function.

```
mouse_rectang:
    cmp     #1,intin+8
    beq     enter_rect
    move    #0,mouse_shape ;arrow
    jsr     #graf_m
    move    #1,mouse_rect
    bra     evnt_mult

enter_rect:
    move    #0,mouse_rect
    move    #4,mouse_shape ; open hand
    jsr     graf_m
    bra     evnt_mult
```

objc_edit

'objc_edit' has three distinct modes which are determined by what is passed to 'intin+6'.

0	ed_start	reserved for future use
1	ed_init	turn cursor on
2	ed_char	display text
3	ed_end	cursor off

This code shows the 'ed_char' mode:

```
* objc_edit
    move.w  index,intin         ; object index
    move.w  key,intin+2         ; value of keyboard press
    move.w  char_pos,intin+4    ; character position
    move.w  #2,intin+6          ; print text
    move.l  #parent,addrin
    move.l  #objc_edit,aespb
    jsr     aes
    move.w  intout+2,char_pos   ; get next character position
```

The first word of 'intout' gives a zero if an error occurred or a positive value otherwise. 'intout+2' gives the next character position of the field.

Basic operation

The basic operation of the program is as follows:

1. Display dialog box.
2. Turn cursor on.
3. Use evnt_multi to monitor events
4. If keyboard check value returned and do various routines. Go back to event_multi.
5. If mouse pointer movement into/out of dialog box alter shape. Go back to event_multi.
6. If mouse pointer click see if over 'OK' button. If it is exit, otherwise go back to event_multi.

Carriage return

The carriage return (ie pressing the Return key) routine (cr:) deserves special attention. It also serves the 'cursor_up' routine and when the cursor down key is pressed it processes this as if a carriage return had been received. This is because pressing the cursor down key can result in a similar action to pressing the Return key– placing the cursor at the end of any present text in the next line or field down. Pressing the cursor up key is also similar as the cursor is placed at the end of any text string in the previous field.

The 'cr:' routine first turns the cursor off and then checks for to see if the cursor up key has been pressed. If it has then the value of 'index' is decremented by one so that the previous editable object is now the one that will be dealt with by the GEM/AES. If the cursor up key has not been pressed then the value of 'index' is incremented by one unless it is the last object. Fortunately, GEM/AES checks to see if any characters are present in the field and puts the cursor at the next editable position, so if any characters are present then the cursor position is passed via 'intout+2'.

The last thing to do is to switch the cursor on.

```
cr:
*Carriage return
        move.w    index,intin         ; object index
        move.w    key,intin+2         ; value
        move.w    char_pos,intin+4    ; position of cursor
        move.w    #3,intin+6          ; cursor off
        move.l    #objc_edit,aespb
        move.l    #parent,addrin
        jsr       aes
        move      intout+2,char_pos   ; store cursor pos

        cmpi.b    #1,curs_up_flag     ; cursor up?
        bne       not_curs_up         ; no
        sub       #1,index            ; yes, go back one index
        bra       dont_add
not_curs_up:
        cmpi.w    #10,index           ; last editable object
        beq       dont_add

        add.w     #1,index            ; do next object
dont_add:

        move.w    index,intin
        move.w    key,intin+2
        move.w    char_pos,intin+4
        move.w    #1,intin+6          ; cursor on
        jsr       aes
        move      intout+2,char_pos
        clr.b     curs_up_flag
        bra       evnt_mult           ; and wait for another key press
```

objc_find

When a mouse click is detected then the routine 'mouse:' is invoked. 'objc_find' takes the 'mx' and 'my' values from 'evnt_multi'. Whatever object the mouse pointer is over the index value is given in 'intout'. If it is the 'OK' button index then an alert box is presented to the user. If it isn't the 'OK' object index then nothing is done.

```
mouse:
*objc_find
```

```
move      #0,intin     ; index of first obj in tree to be searched
move      #1,intin+2   ; depth
move      mx,intin+4   ; x coord of object
move      my,intin+6   ; y coord of object
move.l    #objc_find,aespb
move.l    #parent,addrin
jsr       aes
cmp       #11,intout   ; OK button
beq       do_alert
bra       evnt_mult
```

Alert string

The alert string demonstrates how to use characters not available in the normal manner. The way to do this is to use the ASCII characters after the inverted commas as in the example.

```
alert_string:   dc.b   "[1][ Are you sure you want to|"
                dc.b   " leave !? ",28,29
                dc.b   "|        ",30,31,191, "|You should Save your|"
                dc.b   " work before exiting.][ No! | Exit ]",0
```

Evnt_multi and drop down menus

Using 'evnt__multi' and drop down menus follows a similar process to that described in chapter fourteen, as 'evnt__multi' can also receive messages. As 'evnt__multi' can also accept keyboard presses it is possible to set up drop down menus so that they can seem to respond to key presses such as ALT–L which often selects the menu entry *Load...*, or ALT–Q which selects the *Quit* option. This method of 'hot keys' is used in the supplied text editor and many other ST applications.

Often the symbol for opening a window to its maximum size and reducing it to its prior size is used to represent the ALT key in a drop down menu entry. It is usually placed after the text so that for instance in the menu entry *Quit...*, ALT–Q would be placed as in the zzSoft text editor. Pressing ALT–Q would probably bring an alert box asking if the user really wanted to quit. To get the ALT character in the RCP double click on the menu entry press Control–G in the edit field.

GEM10.S demonstrates the technique of using 'hot keys':

```
* GEM10.S
* Uses drop down menu, uses dialog box, 'objc_change', and
* 'form_dial'. Equates modified from the file EXAMPLE.H
* Resource file EXAMPLE2.RSC
* NOTE: resource file must be in root directory of drive program run
* from. Uses 'evnt_multi' instead of 'evnt_mesag', so that 'hot keys'
* may be used.

* dialog box
tree001     equ   0
cancel      equ   2
button2     equ   4
button1     equ   5
button3     equ   6
date        equ   9
modem       equ   11
printer     equ   12
ok          equ   14

* MENU
tree002     equ   1
desk        equ   3
file        equ   4
page        equ   5
about       equ   8
save_as     equ   18
load        equ   20
quit        equ   22
g_top       equ   24
g_bottom    equ   25
page0       equ   27
etc         equ   28

* a3 is used to store address for AES calls using addrin
* a4 for editable text fields if any

* header
    move.l    a7,a5
    move.l    #ustk,a7
```

```
        move.l      4(a5),a5
        move.l      12(a5),d0
        add.l       20(a5),d0
        add.l       28(a5),d0
        add.l       #$100,d0
        move.l      d0,-(sp)
        move.l      a5,-(sp)
        move        d0,-(sp)
        move        #$4a,-(sp)
        trap        #1
        add.l       #12,sp

* appl_init()
        move.l      #appl_init,aespb
        jsr         aes     ; call AES

        move.l      #rsc_load,aespb     ; AES load a resource file
        move.l      #rsc_file,addrin    ; name of resource file to  be
* loaded
        jsr         aes
        cmpi.w      #0,intout       ; was the resource file loaded
        beq         exit2           ; no

* dialog box
        move.l      #rsc_gaddr,aespb        ; get address of resource tree
        move        #0,intin                ; tree structure
        move        #tree001,intin+2        ; dialog box
        bsr         aes
        cmpi.w      #0,intout   ; error
        beq         exit        ; yes

        move.l      addrout,dialog      ; place address in dialog

* menu
        move.l      #rsc_gaddr,aespb    ; get address of resource tree
        move        #0,intin            ; tree structure
        move        #tree002,intin+2    ; drop down menu
        bsr         aes
        cmpi.w      #0,intout       ; error
        beq         exit            ; yes
```

```
   move.l     addrout,men_bar     ; place address in menu_bar

* put menu bar on screen
   move.l     #menu_bar,aespb  ; display menu object tree
   move.l     men_bar,addrin
   move       #1,intin         ; show menu_bar
   bsr        aes

   bsr        arrow

   jsr        menu_t           ; change menu title to normal video

evnt_mult:
   jsr        menu_t
   move       #1+2+16,intin   ; keyboard, mouse, report
   move       #1,intin+2      ; number of clicks
   move       #1,intin+4      ; left button
   move       #1,intin+6      ; left button down
   move       #0,intin+8      ; mouse enter(0)/leave(1) rectangle
   move       #0,intin+10     ; x coord no coords for #1 rectangle
   move       #0,intin+12     ; y coord #1 rect
   move       #0,intin+14     ; width
   move       #0,intin+16     ; height
   move       #0,intin+18
   move       #0,intin+20     ; no coords for second rect
   move       #0,intin+22
   move       #0,intin+24
   move       #0,intin+26
   move       #0,intin+28     ; timer low word
   move       #0,intin+30     ; timer high word
   move.l     #evnt_multi,aespb       ; display menu object tree
   move.l     #message_buffer,addrin
   jsr        aes
   move.w     intout+2,mx         ; mouse x coord
   move.w     intout+4,my
*  cmpi.w     #2,intout        ; mouse click event
*  beq        mouse

   cmpi.w     #1,intout
   beq        key_board
   cmpi.w     #$10,intout
```

```
    beq         report_mouse

    jsr         do_alert

    bra         evnt_mult

report_mouse:
    cmp         #desk,message_buffer+6  ; get Desk menu bar
    beq         do_menu
    cmp         #file,message_buffer+6  ; get File menu bar
    beq         do_menu
    bra         evnt_mult

key_board:
    move        intout+10,d7        ; get key press
    cmpi.w      #$1000,d7           ; ALT Q
    beq         exit_alert          ; quit
    cmpi.w      #$2600,d7           ; ALT L LOAD
    beq         load_file
    cmpi.w      #$3B00,d7           ; F1
    beq         f1
    cmpi.w      #$3C00,d7           ; F2
    beq         f2
    cmpi.w      #$3D00,d7           ; F3
    beq         f3
    cmpi.w      #$4400,d7           ; F10
    beq         f4
    bra         evnt_mult

do_menu:
    cmp         #about,message_buffer+8
    beq         got_about
    cmp         #quit,message_buffer+8
    beq         exit_alert
    jsr         do_alert
    bra         evnt_mult

load_file:
    jsr         load_alert
    bra         evnt_mult
exit_alert:
```

```
    jsr      quit_alert
    bra      evnt_mult

* F1-F4 unassigned
f1:
f2:
f3:
f4:
    bra      evnt_mult

got_about:
    move     #0,form_flag   ; reserve area of screen memory
    move.l   dialog,a3
    move     #date,d4
    jsr      do_dialog      ; display dialog box and interact with it
    move     #3,form_flag   ; release area of screen memory
    jsr      form_d         ; do it
    bra      evnt_mult

do_dialog:
    bsr      form_center    ; get centred coords of dilaog
* box
    bsr      form_d         ; reserve screen memory
    bsr      obdraw         ; draw it on screen
    bsr      f_do           ; handle interaction with user
    bsr      ob_change      ; reset ok or cancel to non selected
    rts

do_alert:
    move.l   #form_alert,aespb
    move     #1,intin          ; first button
    move.l   #alert_string,addrin
    bsr      aes
    rts

load_alert:
    move.l   #form_alert,aespb
    move     #1,intin          ; first button
    move.l   #load_string,addrin
    bsr      aes
    rts
```

```
quit_alert:
     move.l     #form_alert,aespb
     move       #1,intin           ; first button
     move.l     #exit_string,addrin
     bsr        aes
     cmpi.w     #1,intout
     beq        exit
     rts

ob_change:
     move.l     #objc_change,aespb
     move       intout,intin    ; ok or cancel- from 'form_do'
     move       #0,intin+2
     move       cx,intin+4
     move       cy,intin+6
     move       cw,intin+8
     move       ch,intin+10
     move       #0,intin+12     ; new status- not selected
     move       #1,intin+14     ; not redrawn after status change
     move.l     a3,addrin
     bsr        aes
     rts

menu_t:
     move.l     #menu_tnormal,aespb
     move       message_buffer+6,intin
     move       #1,intin+2
     move.l     men_bar,addrin
     bsr        aes
     rts

obdraw:
     move       #0,intin
     move       #2,intin+2
     move       cx,intin+4
     move       cy,intin+6
     move       cw,intin+8
     move       ch,intin+10
     move.l     a3,addrin
     move.l     #object_draw,aespb
     bsr        aes
```

```
        rts

*       .globl      f_do
* a4 contains editable text field if any
f_do:  move.l     #form_do,aespb      ; form_do
        move        d4,intin            ; editable text field
        move.l      a3,addrin
        bsr         aes
        rts

* form_dial
form_d:
        move        form_flag,intin
        move        cx,intin+2
        move        cy,intin+4
        move        cw,intin+6
        move        ch,intin+8
        move        cx,intin+10
        move        cy,intin+12
        move        cw,intin+14
        move        ch,intin+16
        move.l      #form_dial,aespb
        bsr         aes
        rts

form_center:
        move.l      #f_center,aespb
        move.l      a3,addrin
        bsr         aes
        movem.w     intout+2,d0-d3
        movem.w     d0-d3,cx
        rts

* AES subroutine
aes:   move.l     #aespb,d1
        move.l      #$c8,d0
        trap        #2
        rts

arrow:
* graf_mouse
```

```
        movem.l    a0-a6/d0-d7,-(sp)
        move.l     #graf_mouse,aespb
        move       #0,intin          ; arrow
        bsr        aes
        movem.l    (sp)+,a0-a6/d0-d7
        rts

exit:
        move.l     #rsc_free,aespb     ; release memory taken up by  the
* resource file
        bsr        aes

exit2:

* appl_exit()
        move.l     #appl_exit,aespb
        bsr        aes          ; call AES

        clr.w      -(sp)
        trap       #1

        ds.l   256
ustk:   ds.l   1

aespb:  dc.l   contrl,global,intin,intout,addrin,addrout

object_draw:    dc.w    42,6,1,1,0
form_do:        dc.w    50,1,2,1,0
f_center:       dc.w    54,0,5,1,0
menu_bar:       dc.w    30,1,1,1,0
form_dial:      dc.w    51,9,1,1,0

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
evnt_multi:     dc.w    25,16,7,1,0
rsc_load:       dc.w    110,0,1,1,0
rsc_gaddr:      dc.w    112,2,1,0,1
rsc_free:       dc.w    111,0,1,0,1
graf_mouse:     dc.w    78,1,1,1,0
menu_tnormal:   dc.w    33,2,1,1,0
```

```
objc_change:     dc.w   47,8,1,1,0
form_alert:      dc.w   52,1,1,1,0

message_buffer: ds.b   16

* these 4 must stay together
cx:    ds.w   1
cy:    ds.w   1
cw:    ds.w   1
ch:    ds.w   1

contrl:          ds.w   128
intin:           ds.w   128
intout:          ds.w   128
global:          ds.w   128
addrin:          ds.w   128
addrout:         ds.w   128

dialog:          ds.l   1
men_bar:         ds.l   1
form_flag:       ds.w   1

rsc_file:        dc.b   "example2.rsc",0

alert_string:    dc.b   "[3][There is nothing | assigned to this |"
                 dc.b   " menu selection! | Please try another.][ OK ]",0

load_string:     dc.b   "[3][Load a file | assigned to this |"
                 dc.b   " menu selection!][ OK ]",0
exit_string:     dc.b   "[1][Do you really | want to quit|"
                 dc.b   " from this program!][ OK | NO ]",0
mx:    ds.w   1
my:    ds.w   1
```

Chapter 16
File Selector/Bit Images

This chapter looks at the GEM file selector box and a short example program demonstrates its use. Also, two methods of using bit images in hand constructed dialog boxes is shown.

GEM file selector

The file selector box is a ready made dialog box provided by GEM which facilitates the selection of files from a disk drive. Some find the GEM file selector box to be very limited in design and consequently there are quite a number of excellent substitute selector boxes on the market, from PD to commercial offerings.

These substitute boxes offer many advantages over the flawed GEM one, such as radio buttons for the selection of other drives, automatic display of file size etc. The source code below will load the GEM original from ROM or the substitute ones.

The new STE TOS has an improved file selector box as does TOS 1.4, but many people are stuck with the original or have to use a PD one executed usually from an AUTO folder at start up.

If the mouse pointer is in the file selector box and an underline is typed into the editable line the ST will crash, or at least it will if you have one of the older TOS's. There is no way around this except to use the substitute boxes, although typing an underline at the same time the mouse pointer is in the box must be a fairly rare occurrence.

Note that the screen or a window has to be refreshed after a fsel_input call as GEM does not clear the screen itself. You have to do it your self. So in the example below if we did not exit but carried on then the file selector box would be left on the screen and would overwrite anything we had prior to its display. Solution to this problem is to save the screen in a buffer prior to the call and restore it afterwards.

Also, the VDI clipping rectangle is altered by the appearance of the file selector box, and it is up to the programmer to set the any clipping after use of the file selector box.

```
* GEM11.S
* This program illustrates the use of the GEM file selector box
* by use of the AES call 'fsel_input'

* header
	move.l	a7,a5
	move.l	#ustk,a7
	move.l	4(a5),a5
	move.l	12(a5),d0
	add.l	20(a5),d0
	add.l	28(a5),d0
	add.l	#$100,d0
	move.l	d0,-(sp)
	move.l	a5,-(sp)
	move	d0,-(sp)
	move	#$4a,-(sp)
	trap	#1
	add.l	#12,sp

* appl_init()
	move.l	#appl_init,aespb
	jsr	aes	; call AES

	move.w	#$19,-(sp)	; Get current drive
	trap	#1
	addq.l	#2,sp
	add.b	#65,d0	; alter from number to letter
	move.b	d0,ddir

	move.l	#fsel_input,aespb
	move.l	#ddir,addrin	; initial directory and
* drive to be dislayed
	move.l	#fsel_file,addrin+4	; initial file selection
* to be displayed
	jsr	aes
	bra	exit

aes:	move.l	#aespb,d1
```

```
        move.l      #$c8,d0
        trap        #2
        rts
exit:
* appl_exit()
        move.l      #appl_exit,aespb
        bsr         aes         ; call AES

        clr.w       -(sp)
        trap        #1


        ds.l    256

ustk:   ds.l    1

contrl: ds.w    12
intin:  ds.w    128
intout: ds.w    128
global: ds.w    16

addrin:         ds.w    128
addrout:        ds.w    128

aespb:  dc.l    contrl,global,intin,intout,addrin,addrout

fsel_input:     dc.w    90,0,2,2,0
appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0

fsel_file:      ds.w    8

ddir:   dc.b    "A:\*. S"

        ds.b    56
```

'fsel_input' expects two addresses to be passed to the first long of addrin and addrin+4. The first parameter is the address of the buffer that holds the path of the directory that is initially displayed, eg C:\AUTO\NEODESK.PRG. Wildcards can be used eg '*.*' for all files, C:*.PRG for all program files, etc. The actual path selected, including drive is returned in the same buffer at exit from 'fsel_input'. Note that the buffer should be about 56 bytes in size to accommodate any path size. The second is the address of the buffer that holds the string that specifies the actual choice of file. In the example above the choice has been left out. The choice of the user, if any, is returned in the same buffer.

'intout' also returns some results. The first word of 'intout' should contain either zero for an error occurred, or a number greater than zero for 'Ok'. 'intout+2' contains either a zero or a one, zero signifying cancel was selected, and a one 'Ok' was selected.

Bit images in dialog boxes

As the supplied RCP does not support the use of bit images in dialog boxes the rest of this chapter is devoted to this subject. Two methods are shown one from first principles, and the other using a bit image from the DEGAS ELITE art program.

```
* GEM12.S
* This program displays a dialog box and bit mapped image. Both
* are constructed from first principles. It cannot be assembled by
* zzSoft's assembler. See GEM13.S for same program but
* converted for use with zzSoft's assembler

* header
        move.l      a7,a5
        move.l      #ustk,a7
        move.l      4(a5),a5
        move.l      12(a5),d0
        add.l       20(a5),d0
        add.l       28(a5),d0
        add.l       #$100,d0
        move.l      d0,-(sp)
        move.l      a5,-(sp)
        clr         -(sp)
        move        #$4a,-(sp)
        trap        #1
        add.l       #12,sp
* appl_init()
```

```
    move.l    #appl_init,aespb
    jsr       aes       ; call AES

* get current screen res
    move      #4,-(sp)
    trap      #14
    addq.l    #2,sp
* res returned in d0

    cmp       #2,d0                 ; is it high res
    bne       dont_alter_coords     ; no

    move.l    #parent,a5            ; address of tree in a5
    move.l    #9,d5                 ; number of objects
    bsr       alter_coords

dont_alter_coords:
    bsr       f_center
    bsr       obdraw
    bsr       f_do
    bra       exit

obdraw:
    move      #0,intin
    move      #2,intin+2
    move      d0,intin+4
    move      d1,intin+6
    move      d2,intin+8
    move      d3,intin+10
    move.l    #parent,addrin
    move.l    #object_draw,aespb
    bsr       aes
    rts

f_do: move.l  #form_do,aespb
    clr.w     intin         ; no editable text field
    move.l    #parent,addrin
    bsr       aes
    rts

f_center:
```

```
        move.l    #form_center,aespb
        move.l    #parent,addrin
        jsr       aes
        movem.w   intout+2,d0-d3
        rts

alter_coords:
* adjust object data for high res screen
        add.l     #18,a5
        move.w    (a5),d3
        mulu.w    #2,d3
        move      d3,(a5)+
        add.l     #2,a5
        move      (a5),d3
        mulu.w    #2,d3
        move      d3,(a5)+
        dbf       d5,alter_coords
        rts

* AES subroutine
aes:    move.l    #aespb,d1
        move.l    #$c8,d0
        trap      #2
        rts

exit:
* appl_exit()
        move.l    #appl_exit,aespb
        bsr       aes        ; call AES

        clr.w     -(sp)
        trap      #1

        ds.l   256
ustk:   ds.l   1

tl:     dc.l   t_1
        dc.w   4,16,0,0,$01f1

t_1:    dc.l   %00000000000000000000000000000000
```

```
        dc.l    %00000000000000000000000000000000
        dc.l    %11111111111111111111111111111111
        dc.l    %11111111111111111111111111111111
        dc.l    %00000011110000000000000000000000
        dc.l    %00000011110000000111111110000000
        dc.l    %00000011110000001111111110000000
        dc.l    %00000011110000001110000111000000
        dc.l    %00000011110000001110000111000000
        dc.l    %00000011110000001111111110000000
        dc.l    %00000011110000001111111110000000
        dc.l    %00000011110000001110001111000000
        dc.l    %00000011110000001110000111100000
        dc.l    %00000011110000001110000011110000
        dc.l    %00000000000000000000000011110000
        dc.l    %11111111111111111111111111111110

t2:     dc.l    t_2
        dc.w    4,16,0,0,$01f1

t_2:    dc.l    %00000000000000000000000000000000
        dc.l    %00000000000000000000000000000000
        dc.l    %11111111111111111111111111111111
        dc.l    %11111111111111111111111111111111
        dc.l    %00000000000000000000000000000000
        dc.l    %00111111110000001111111100000000
        dc.l    %01111111111000011111111110000000
        dc.l    %01110000111000011100001110000000
        dc.l    %01110000111000011100001110000000
        dc.l    %01111111111000011100001110000000
        dc.l    %01111111111000011100001110000000
        dc.l    %01110000111000011100001110000000
        dc.l    %01110000111000011111111110000000
        dc.l    %01110000111000011111111100000000
        dc.l    %00000000000000000000000000000000
        dc.l    %11111111111111111111111111111111

t3:     dc.l    t_3
        dc.w    4,16,0,0,$01f1

t_3:    dc.l    %00000000000000000000000000000000
        dc.l    %00000000000000000000000000000000
```

```
        dc.l    %11111111111111111111111111111111
        dc.l    %11111111111111111111111111111111
        dc.l    %00000000000000000000000000000000
        dc.l    %01111111111000011111111000000000
        dc.l    %01111111111000011111111110000000
        dc.l    %01110000000000011100001110000000
        dc.l    %01110000000000011100001110000000
        dc.l    %01111111111000011111111110000000
        dc.l    %01111111111000011111111110000000
        dc.l    %01110000000000011100011110000000
        dc.l    %01111111111000011100001111000000
        dc.l    %01111111111000011100000111100000
        dc.l    %00000000000000000000000011110000
        dc.l    %11111111111111111111111111111111

t4:     dc.l    t_4
        dc.w    4,16,0,0,$01f1

t_4:    dc.l    %00000000000000000000000000000000
        dc.l    %00000000000000000000000000000000
        dc.l    %11111111111111111111111111111111
        dc.l    %11111111111111111111111111111111
        dc.l    %00000000000000000000000000000000
        dc.l    %01111111110000011111111111100000
        dc.l    %01111111110000011111111111100000
        dc.l    %01111000000000000000111100000000
        dc.l    %00011110000000000000111100000000
        dc.l    %00000011100000000000111100000000
        dc.l    %00000011110000000000111100000000
        dc.l    %00000011110000000000111100000000
        dc.l    %00111111110000000001111000111100
        dc.l    %00111111110000000001111000111100
        dc.l    %00000000000000000000000000000000
        dc.l    %11111111111111111111111111111111

title1: dc.b    ' Integrated Accounts',0

title2: dc.b    ' Software',0

t6:     dc.l    ty,null,null
        dc.w    3,0,2,$13b2,0,1,14,0  ; gives red background, blue text
```

```
ty:     dc.b  'Version: 1.00',0

t7:     dc.b  189,' Someones Software',191,' 1990',0

null:   dc.b  0

exit_:  dc.l  text_ok,null,null
        dc.w  3,0,2,$1202,0,3,5,0

text_ok:        dc.b  'OK',32,175,0

aespb:  dc.l  contrl,global,intin,intout,addrin,addrout

object_draw:    dc.w   42,6,1,1,0
form_do:        dc.w   50,1,2,1,0

parent:         dc.w   -1,1,9,20,0,16        ; large box
        dc.l    $22020
        dc.w    170,50,250,120

        dc.w    2,-1,-1,28,0,0
        dc.l    title1
        dc.w    35,30,90,15

        dc.w    3,-1,-1,23,0,0              ; 23=bitblk
        dc.l    t1
        dc.w    10,10,16,19

        dc.w    4,-1,-1,23,0,0
        dc.l    t2
        dc.w    40,10,16,19

        dc.w    5,-1,-1,23,0,0
        dc.l    t3
        dc.w    70,10,16,19

        dc.w    6,-1,-1,23,0,0
        dc.l    t4
        dc.w    100,10,16,19
        dc.w    7,-1,-1,28,0,0
        dc.l    title2
```

```
        dc.w    70,40,90,15

        dc.w    8,-1,-1,22,7,0
        dc.l    exit_           ; exit
        dc.w    100,100,50,15

        dc.w    9,-1,-1,22,0,0
        dc.l    t6              ; version
        dc.w    50,60,150,15

        dc.w    0,-1,-1,28,32,0
        dc.l    t7              ; (c) copyright
        dc.w    30,80,150,15

* GEM arrays

contrl:         ds.w    12
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    16
addrin:         ds.w    128
addrout:        ds.w    128

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
form_center:    dc.w    54,0,5,1,0
```

The 'bitblk' structure is in the form:

word	name	
0 and 1	bi_pdata	a pointer to a bit mapped array
2	bi_wp	width of bit map in bytes
3	bi_hl	height of bit map in pixels
4	bi_x	x coord
5	bi_y	y coord
6	bi_color	colour of graphic

'bi_wb'– this number must be even, and 'bi_color' does not seem to have any effect. The colour is always black and white.

If we look at the first tedinfo structure we can see that the structure conditions are fulfilled:

```
t1:    dc.l   t_1         ; pointer to bit mapped array
       dc.w   4,16,0,0,$01f1     ; 4 bytes wide (one long word)
* 16 pixels height, 0- X coord, 0- Y coord, $01f1 colour data
```

The bit mapped array is easy to construct as it can be done by hand. Unfortunately it really needs two arrays one for medium resolution, and one for high res as there is a height discrepancy in displaying the bit image. The rest of the program should be easy enough to follow.

Also, unfortunately the zzSoft assembler cannot accept the 'dc.l %0000000000' binary format. It must first be converted to hexadecimal representation. Other assemblers can accept data in binary format such as Devpac. Taking the first 4 lines the conversion would be like this:

```
t_1:   dc.l   %00000000000000000000000000000000
       dc.l   %00000000000000000000000000000000
       dc.l   %11111111111111111111111111111111
       dc.l   %11111111111111111111111111111111
```

Converting

```
t_1:   dc.l   $00000000
       dc.l   $00000000
       dc.l   $ffffffff
       dc.l   $ffffffff
```

As each long (dc.l) is 32 bits then there must be 4 bytes of information in each line which then can be translated as above.

GEM13.S gives the correct file for assembling with the supplied assembler. The code in all respects is the same except the binary bit mapped notation has been changed to hexadecimal notation.

```
* GEM13.S
* This program displays a dialog box with a bit image in it.
* The dialog box and bit mapped image are both constructed from
* first principles.
```

```
* header
        move.l    a7,a5
        move.l    #ustk,a7
        move.l    4(a5),a5
        move.l    12(a5),d0
        add.l     20(a5),d0
        add.l     28(a5),d0
        add.l     #$100,d0
        move.l    d0,-(sp)
        move.l    a5,-(sp)
        clr       -(sp)
        move      #$4a,-(sp)
        trap      #1
        add.l     #12,sp

* appl_init()
        move.l    #appl_init,aespb
        jsr       aes      ; call AES

* get current screen res
        move      #4,-(sp)
        trap      #14
        addq.l    #2,sp
* res returned in d0

        cmp       #2,d0               ; is it high res
        bne       dont_alter_coords   ; no

        move.l    #parent,a5          ; address of tree in a5
        move.l    #9,d5               ; number of objects
        bsr       alter_coords

dont_alter_coords:
        bsr       f_center
        bsr       obdraw
        bsr       f_do
        bra       exit

obdraw:
        move      #0,intin
        move      #2,intin+2
```

```
        move      d0,intin+4
        move      d1,intin+6
        move      d2,intin+8
        move      d3,intin+10
        move.l    #parent,addrin
        move.l    #object_draw,aespb
        bsr       aes
        rts

f_do:   move.l    #form_do,aespb
        clr.w     intin      ; no editable text field
        move.l    #parent,addrin
        bsr       aes
        rts

f_center:
        move.l    #form_center,aespb
        move.l    #parent,addrin
        jsr       aes
        movem.w   intout+2,d0-d3
        rts

alter_coords:
* adjust object data for high res screen
        add.l     #18,a5
        move.w    (a5),d3
        mulu.w    #2,d3
        move      d3,(a5)+
        add.l     #2,a5
        move      (a5),d3
        mulu.w    #2,d3
        move      d3,(a5)+
        dbf       d5,alter_coords
        rts

* AES subroutine
aes:    move.l    #aespb,d1
        move.l    #$c8,d0
        trap      #2
        rts
```

```
exit:
* appl_exit()
        move.l      #appl_exit,aespb
        bsr         aes         ; call AES

        clr.w       -(sp)
        trap        #1

        ds.l    256
ustk:   ds.l    1

t1:     dc.l    t_1
        dc.w    4,16,0,0,$01f1

t_1:    dc.l        $00000000
        dc.l        $00000000
        dc.l        $ffffffff
        dc.l        $ffffffff
        dc.l        $03c00000
        dc.l        $03c0ff00
        dc.l        $03c1ff80
        dc.l        $03c1c380
        dc.l        $03c1c380
        dc.l        $03c1ff80
        dc.l        $03c1ff80
        dc.l        $03c1c780
        dc.l        $03c1c3c0
        dc.l        $03c1c1e0
        dc.l        $000000f0
        dc.l        $ffffffff

t2:     dc.l        t_2
        dc.w        4,16,0,0,$01f1

t_2:    dc.l        $00000000
        dc.l        $00000000
        dc.l        $ffffffff
        dc.l        $ffffffff
        dc.l        $00000000
```

```
        dc.l        $3fc1ff00
        dc.l        $7fe1ff80
        dc.l        $70e1c380
        dc.l        $70e1c380
        dc.l        $7fe1c380
        dc.l        $7fe1c380
        dc.l        $70e1c380
        dc.l        $70e1ff80
        dc.l        $70e1ff00
        dc.l        $00000000
        dc.l        $ffffffff

t3:     dc.l        t_3
        dc.w        4,16,0,0,$01f1

t_3:    dc.l        $00000000
        dc.l        $00000000
        dc.l        $ffffffff
        dc.l        $ffffffff
        dc.l        $00000000
        dc.l        $7fe1fe00
        dc.l        $7fe1ff80
        dc.l        $7001c380
        dc.l        $7001c380
        dc.l        $7fe1ff80
        dc.l        $7fe1ff80
        dc.l        $7001c780
        dc.l        $7fe1c3c0
        dc.l        $7fe1c1e0
        dc.l        $000000f0
        dc.l        $ffffffff

t4:     dc.l        t_4
        dc.w        4,16,0,0,$01f1

t_4:    dc.l        $00000000
        dc.l        $00000000
        dc.l        $ffffffff
        dc.l        $ffffffff
        dc.l        $00000000
        dc.l        $7fc1ffe0
```

```
        dc.l        $7fc1ffe0
        dc.l        $78001e00
        dc.l        $1e001e00
        dc.l        $03801e00
        dc.l        $03c01e00
        dc.l        $03c01e00
        dc.l        $3fc01e3c
        dc.l        $3fc01e3c
        dc.l        $00000000
        dc.l        $ffffffff

title1:             dc.b   ' Integrated Accounts',0

title2:             dc.b   ' Software',0

t6:     dc.l   ty,null,null
        dc.w    3,0,2,$13b2,0,1,14,0  ; gives red background, blue text

ty:     dc.b   'Version: 1.00',0

t7:     dc.b   189,' Someones Software',191,' 1990',0

null:   dc.b   0

exit_:  dc.l   text_ok,null,null
        dc.w    3,0,2,$1202,0,3,5,0

text_ok:            dc.b   'OK',32,175,0

aespb:  dc.l   contrl,global,intin,intout,addrin,addrout

object_draw:        dc.w   42,6,1,1,0

form_do:            dc.w   50,1,2,1,0

parent:             dc.w   -1,1,9,20,0,16        ; large box
        dc.l        $22020
        dc.w        170,50,250,120
        dc.w        2,-1,-1,28,0,0
        dc.l        title1
```

```
        dc.w        35,30,90,15

        dc.w        3,-1,-1,23,0,0        ; 23=bitblk;
        dc.l        t1
        dc.w        10,10,16,19

        dc.w        4,-1,-1,23,0,0
        dc.l        t2
        dc.w        40,10,16,19

        dc.w        5,-1,-1,23,0,0
        dc.l        t3
        dc.w        70,10,16,19

        dc.w        6,-1,-1,23,0,0
        dc.l        t4
        dc.w        100,10,16,19

        dc.w        7,-1,-1,28,0,0
        dc.l        title2
        dc.w        70,40,90,15

        dc.w        8,-1,-1,22,7,0
        dc.l        exit_        ; exit
        dc.w        100,100,50,15

        dc.w        9,-1,-1,22,0,0
        dc.l        t6           ; version
        dc.w        50,60,150,15

        dc.w        0,-1,-1,28,32,0
        dc.l        t7           ; (c) copyright
        dc.w        30,80,150,15

* GEM arrays

contrl:         ds.w    12
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    16
addrin:         ds.w    128
```

```
addrout:        ds.w   128

appl_init:      dc.w   10,0,1,0,0
appl_exit:      dc.w   19,0,1,0,0
form_center:    dc.w   54,0,5,1,0
```

Fortunately it is not necessary to have to tediously convert the bit mapped image to hex, nor to create the image by hand as it is possible to use DEGAS ELITE to create the image.

Using DEGAS

After loading DEGAS ELITE the *Block* drop down menu should be selected and from this the *Format* entry should be selected. The *Icon* file format should be clicked on.

Next the particular image that is wanted should be drawn. When a satisfactory drawing is obtained by one's own efforts or by importing someone else's work and modifying, the *Block* option should be clicked on from the menu screen. Going back to the drawing screen press ESC to get two large crossed lines. Position the cross–lines at the top left of your drawing and holding down the left mouse button draw a rectangle around your drawing. As soon as the block is cut out return to the menu screen and save the block as an '.ICN' file. This file can now be converted from its present C structure form to a format suitable for inclusion in assembly language source code as shown in GEM14.S

So that the reader can see the original 'ICN' file called 'EXAMPLE.ICN' before inclusion into GEM14.S this file may be found on the supplied disk.

First the '.ICN' file must be converted for inclusion in our source code. The following procedure will convert it correctly:

1. Load the 'ICN' file into zzSoft's text editor and

2. Delete any '{', and '}' and place '*' in front of the DEGAS ELITE definitions and text. See example below.

3. Replace ' 0x' with '$' using the *Replace All* option. Position cursor

at start of bit mapped block before clicking on *Replace All.*

At end of replacement press ALT–T to go back to the top of the file. Do not put inverted commas (') in *Find (and Replace)* dialog box.

4. Globally replace ', ' with nothing ie the *Replace* field should be empty. This gets rid of the end comma.

5. All that needs doing now is to put a 'dc.w' in front of the hex data. This is done by globally replacing ' ' with ' dc.w ', and the file is ready.

6. Save file with a '.S' extension and insert into your assembly language source code with the 'insert file' option in the text editor by pressing ALT–I.

```
* GEM14.S
* This program displays a dialog box with a bit mapped
* image, taken from a DEGAS ELITE icon block file.

* header
        move.l      a7,a5
        move.l      #ustk,a7
        move.l      4(a5),a5
        move.l      12(a5),d0
        add.l       20(a5),d0
        add.l       28(a5),d0
        add.l       #$100,d0
        move.l      d0,-(sp)
        move.l      a5,-(sp)
        clr         -(sp)
        move        #$4a,-(sp)
        trap        #1
        add.l       #12,sp

* appl_init()
        move.l      #appl_init,aespb
        jsr         aes         ; call AES

* get current screen res
        move        #4,-(sp)
```

```
    trap      #14
    addq.l    #2,sp
* res returned in d0

    cmp       #2,d0                  ; is it high res
    bne       dont_alter_coords      ; no

    move.l    #parent,a5             ; address of tree in a5
    move.l    #2,d5                  ; number of objects
    bsr       alter_coords

dont_alter_coords:
    bsr       f_center
    bsr       obdraw
    bsr       f_do
    bra       exit

obdraw:
    move      #0,intin
    move      #2,intin+2
    move      d0,intin+4
    move      d1,intin+6
    move      d2,intin+8
    move      d3,intin+10
    move.l    #parent,addrin
    move.l    #object_draw,aespb
    bsr       aes
    rts

f_do: move.l  #form_do,aespb
    clr.w     intin ; no editable text field
    move.l    #parent,addrin
    bsr       aes
    rts

f_center:
    move.l    #form_center,aespb
    move.l    #parent,addrin
    jsr       aes
    movem.w   intout+2,d0-d3
    rts
```

```
alter_coords:
* adjust object data for high res screen
    add.l       #18,a5
    move.w      (a5),d3
    mulu.w      #2,d3
    move        d3,(a5)+
    add.l       #2,a5
    move        (a5),d3
    mulu.w      #2,d3
    move        d3,(a5)+
    dbf         d5,alter_coords
    rts

* AES subroutine
aes:  move.l    #aespb,d1
    move.l      #$c8,d0
    trap        #2
    rts

exit:
* appl_exit()
    move.l      #appl_exit,aespb
    bsr         aes          ; call AES

    clr.w       -(sp)
    trap        #1

      ds.l  256
ustk: ds.l  1

b_map:
    dc.l        bit_map
    dc.w        24,$79,0,0,$22a3,0

bit_map:

* /* DEGAS Elite Icon Definition */
* #define ICON_W 0x00B9     divide by 8 to get number of bytes,
* round up to get even number
* #define ICON_H 0x0079     need this as it is
* #define ICONSIZE 0x05AC   not needed
```

```
* int image[ICONSIZE] =      not needed

    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0100
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0820,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $0000,$0000,$0000,$0000
    dc.w    $001F,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFC0,$0000
    dc.w    $003F,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFE0,$0000
    dc.w    $007F,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFF0,$0000
    dc.w    $007F,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFF0,$0000
    dc.w    $007F,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFFF,$FFFF
    dc.w    $FFFF,$FFFF,$FFF0,$0000
```

```
dc.w    $007F,$FFFF,$FFFF,$FFFF
dc.w    $FFFF,$FFFF,$FFFF,$FFFF
dc.w    $FFFF,$FFFF,$FFF0,$0000
dc.w    $007E,$0000,$0000,$0400
dc.w    $4010,$0010,$0000,$0000
dc.w    $0080,$0000,$03F0,$0000
dc.w    $007E,$0000,$0804,$8000
dc.w    $0000,$4000,$0203,$0000
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$0000,$0000,$0000
dc.w    $0002,$0000,$0000,$0200
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$8000,$0100,$0000
dc.w    $0000,$0000,$0020,$2000
dc.w    $0000,$1120,$03F0,$0000
dc.w    $007F,$0020,$0020,$0000
dc.w    $0002,$2000,$0000,$0000
dc.w    $0008,$0100,$03F0,$0000
dc.w    $007E,$0400,$0000,$0200
dc.w    $0000,$0300,$0040,$0000
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$1100,$0000,$0010
dc.w    $0000,$0004,$0000,$0000
dc.w    $0008,$0000,$03F0,$0000
dc.w    $007E,$0800,$0000,$0000
dc.w    $0000,$0002,$0400,$0000
dc.w    $0000,$0400,$03F0,$0000
dc.w    $007E,$0800,$0000,$0000
dc.w    $0000,$0000,$0000,$0000
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$8000,$0001,$0000
dc.w    $0000,$0000,$0010,$0000
dc.w    $0000,$0888,$03F0,$0000
dc.w    $007E,$4000,$0018,$0000
dc.w    $0400,$0000,$0000,$0001
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$0000,$0210,$0000
dc.w    $0000,$0000,$0000,$0000
dc.w    $0000,$1040,$03F0,$0000
dc.w    $007E,$001F,$FFE7,$FFF8
dc.w    $1FE1,$FE7F,$9FFF,$87F8
```

```
dc.w    $01FF,$FE80,$03F0,$0000
dc.w    $007E,$0010,$0024,$0C08
dc.w    $1021,$0240,$9000,$8408
dc.w    $0100,$0340,$03F0,$0000
dc.w    $007E,$0010,$1024,$0C08
dc.w    $7039,$03C0,$9000,$E408
dc.w    $0100,$0300,$03F0,$0000
dc.w    $007E,$0010,$3FE4,$0C08
dc.w    $4009,$0000,$9030,$2408
dc.w    $0103,$FE20,$03F0,$0000
dc.w    $007E,$0010,$600C,$0C09
dc.w    $C00F,$0000,$9030,$2408
dc.w    $0102,$0000,$03F0,$0000
dc.w    $007E,$0010,$2207,$0039
dc.w    $8343,$0000,$9030,$2408
dc.w    $0102,$0020,$03F0,$0000
dc.w    $007E,$0010,$3F81,$0021
dc.w    $030B,$0000,$9030,$2408
dc.w    $0103,$FB00,$03F0,$0000
dc.w    $007E,$0011,$0C81,$C0E1
dc.w    $0303,$0000,$9500,$E408
dc.w    $0100,$0800,$03F0,$0000
dc.w    $007E,$0018,$0081,$C0E1
dc.w    $0303,$0000,$9000,$8408
dc.w    $0100,$1800,$03F0,$0000
dc.w    $007E,$0032,$3F81,$0021
dc.w    $0003,$03C0,$903F,$8408
dc.w    $0103,$F810,$03F0,$0000
dc.w    $007E,$0050,$2087,$0039
dc.w    $0003,$0A40,$9020,$0428
dc.w    $0102,$0020,$03F0,$0000
dc.w    $007E,$0018,$2004,$0C09
dc.w    $0303,$0240,$9020,$0408
dc.w    $0102,$0400,$03F0,$0000
dc.w    $007E,$0012,$3FE4,$0C09
dc.w    $0303,$0240,$9020,$040F
dc.w    $F907,$FE00,$03F0,$0000
dc.w    $007E,$0012,$0024,$0C09
dc.w    $0303,$0240,$9020,$0400
dc.w    $0900,$0200,$03F0,$0000
dc.w    $007E,$0C30,$0024,$0C29
```

```
dc.w    $0303,$0248,$9020,$0400
dc.w    $0900,$0200,$03F0,$0000
dc.w    $007E,$101F,$FFE7,$FFF9
dc.w    $FFFF,$FE7F,$9FE0,$07FF
dc.w    $F9FF,$FE00,$03F0,$0000
dc.w    $007E,$1204,$4000,$0000
dc.w    $0000,$0000,$0000,$0000
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$2810,$0000,$0001
dc.w    $FFF8,$1FFE,$1FFF,$E001
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$0000,$0000,$0001
dc.w    $0008,$1002,$1000,$2000
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$0000,$0000,$0001
dc.w    $000E,$1002,$1000,$2000
dc.w    $0000,$1484,$03F0,$0000
dc.w    $007E,$0000,$0000,$0001
dc.w    $0302,$1C0E,$1F03,$E000
dc.w    $0000,$0080,$03F0,$0000
dc.w    $007E,$0088,$0000,$0001
dc.w    $0302,$0408,$0102,$0000
dc.w    $0010,$0040,$07F0,$0000
dc.w    $007E,$2000,$0000,$0001
dc.w    $0302,$0418,$0102,$0000
dc.w    $8400,$0130,$03F0,$0000
dc.w    $007E,$0220,$0000,$0001
dc.w    $0302,$0408,$0142,$0000
dc.w    $0000,$2100,$03F0,$0000
dc.w    $007E,$0000,$0000,$0001
dc.w    $008E,$0408,$0102,$0000
dc.w    $0000,$0210,$03F0,$0000
dc.w    $007E,$0000,$0420,$0003
dc.w    $020E,$0408,$0102,$0000
dc.w    $4010,$0030,$03F0,$0000
dc.w    $007E,$0200,$0000,$0001
dc.w    $0302,$0408,$0102,$0000
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$0100,$8800,$0001
dc.w    $0302,$0408,$0102,$0000
dc.w    $0008,$0080,$03F0,$0000
```

```
dc.w    $007E,$0000,$0000,$0001
dc.w    $0302,$1408,$0102,$4000
dc.w    $0005,$0080,$03F0,$0000
dc.w    $007E,$0000,$0000,$0001
dc.w    $2302,$1C4E,$0102,$8000
dc.w    $0008,$0200,$03F0,$0000
dc.w    $007E,$8040,$0000,$2001
dc.w    $000E,$5002,$0106,$0401
dc.w    $0011,$0000,$03F0,$0000
dc.w    $007E,$4900,$0080,$1809
dc.w    $0008,$1002,$0102,$0000
dc.w    $0020,$0400,$03F0,$0000
dc.w    $007E,$0000,$801C,$0003
dc.w    $FFF8,$1FFE,$91FE,$4000
dc.w    $0000,$4804,$03F0,$0000
dc.w    $007E,$0000,$2000,$0000
dc.w    $0000,$0000,$0000,$0002
dc.w    $0010,$0002,$03F0,$0000
dc.w    $007E,$007F,$9FE0,$7F81
dc.w    $FFF8,$7FFF,$1FFF,$E7FF
dc.w    $8000,$0000,$03F2,$0000
dc.w    $007E,$0450,$D020,$4081
dc.w    $0008,$4002,$1000,$3400
dc.w    $9000,$0000,$03F0,$0000
dc.w    $007E,$0040,$F225,$C0E1
dc.w    $001E,$4003,$9000,$2400
dc.w    $E080,$0000,$03F8,$0000
dc.w    $007E,$0040,$0021,$0021
dc.w    $0302,$40C0,$903F,$E400
dc.w    $2009,$0402,$03F0,$0000
dc.w    $007E,$0040,$0027,$0039
dc.w    $0302,$40C0,$9820,$0400
dc.w    $7800,$0000,$03F0,$0000
dc.w    $007E,$C040,$0424,$0C09
dc.w    $0302,$40C0,$9020,$040C
dc.w    $0800,$0000,$03F0,$0000
dc.w    $007E,$0040,$0024,$0C09
dc.w    $0302,$40C0,$B03F,$840C
dc.w    $09FF,$FE00,$03F0,$0000
dc.w    $007E,$0040,$0024,$0C09
dc.w    $000E,$4047,$9000,$840C
```

```
dc.w    $0900,$4200,$03F0,$0000
dc.w    $007E,$1060,$0024,$0C09
dc.w    $0008,$4002,$1000,$840C
dc.w    $0900,$0240,$03F0,$0000
dc.w    $007E,$0148,$F024,$0009
dc.w    $03F8,$40FE,$103F,$840C
dc.w    $09FF,$FE00,$03F0,$0000
dc.w    $007E,$1048,$9024,$0009
dc.w    $0200,$4080,$1020,$040C
dc.w    $0800,$0800,$03F0,$0000
dc.w    $007E,$1040,$9024,$0C09
dc.w    $0200,$4080,$1020,$0400
dc.w    $3804,$4000,$03F2,$0000
dc.w    $007E,$8850,$9024,$0C09
dc.w    $0200,$4080,$103F,$E400
dc.w    $2000,$0200,$03F4,$0000
dc.w    $007E,$2050,$9024,$0C09
dc.w    $0200,$4080,$1000,$2400
dc.w    $E000,$0000,$03F0,$0000
dc.w    $007E,$0140,$9024,$0C09
dc.w    $0200,$4080,$1000,$2400
dc.w    $8000,$8000,$03F0,$0000
dc.w    $007E,$407F,$9FE7,$FFF9
dc.w    $FE00,$7F80,$1FFF,$E7FF
dc.w    $8000,$0300,$03F8,$0000
dc.w    $007E,$0040,$0000,$0000
dc.w    $0000,$0000,$0000,$0000
dc.w    $0001,$2000,$03F0,$0000
dc.w    $007E,$0000,$0001,$FFE7
dc.w    $F9FE,$07F8,$07FF,$E7FF
dc.w    $F800,$0000,$03F0,$0000
dc.w    $007E,$0000,$0001,$0025
dc.w    $090A,$0408,$0400,$2400
dc.w    $0800,$0044,$03F0,$0000
dc.w    $007E,$1000,$0011,$0024
dc.w    $0F02,$1C0E,$1C00,$2400
dc.w    $0800,$0080,$03F0,$0000
dc.w    $007E,$1080,$1001,$C0E4
dc.w    $0002,$1002,$103F,$EC0F
dc.w    $F800,$00A0,$03F0,$0000
dc.w    $007E,$0000,$0000,$4084
```

```
dc.w    $0082,$7803,$9020,$0408
dc.w    $0000,$4000,$03F0,$0000
dc.w    $007E,$0008,$0010,$4886
dc.w    $0002,$42C0,$9020,$0408
dc.w    $0080,$0000,$03F0,$0000
dc.w    $007E,$0000,$0520,$4084
dc.w    $8002,$40C0,$903F,$E40F
dc.w    $E000,$0000,$03F0,$0000
dc.w    $007E,$0200,$0000,$4084
dc.w    $0002,$40C0,$B000,$2400
dc.w    $2100,$4808,$03F0,$0000
dc.w    $007E,$8000,$0080,$4084
dc.w    $0002,$42C0,$9000,$2400
dc.w    $2000,$0080,$03F0,$0000
dc.w    $007E,$4400,$2000,$4084
dc.w    $0F02,$4000,$9030,$240F
dc.w    $E000,$0080,$03F0,$0000
dc.w    $007F,$C081,$1101,$4084
dc.w    $0902,$4000,$9030,$2408
dc.w    $1000,$0400,$03F0,$0000
dc.w    $007E,$2180,$080B,$408C
dc.w    $0902,$40C0,$9038,$2408
dc.w    $0000,$1040,$03F0,$0000
dc.w    $007E,$0C80,$0005,$C0E4
dc.w    $0902,$40C0,$9030,$240F
dc.w    $F800,$1000,$03F0,$0000
dc.w    $007E,$0000,$0081,$0024
dc.w    $0902,$40C0,$9C00,$2400
dc.w    $0800,$0480,$03F0,$0000
dc.w    $007E,$0280,$0101,$00A4
dc.w    $0902,$40C0,$8400,$2500
dc.w    $0840,$00C0,$03F0,$0000
dc.w    $007E,$0802,$6521,$FFE7
dc.w    $F9FE,$7FFF,$87FF,$E7FF
dc.w    $F800,$00A1,$03F0,$0000
dc.w    $007E,$0001,$9C44,$0400
dc.w    $0000,$0000,$0200,$0010
dc.w    $2000,$1100,$03F0,$0000
dc.w    $007F,$0000,$DB88,$0000
dc.w    $1010,$0000,$0000,$0000
dc.w    $0080,$0202,$03F0,$0000
```

```
dc.w    $007F,$8000,$0380,$0000
dc.w    $0004,$0000,$0000,$0000
dc.w    $0080,$0000,$03F0,$0000
dc.w    $007E,$0006,$5200,$0805
dc.w    $2008,$0000,$0000,$0800
dc.w    $0000,$0008,$03F0,$0000
dc.w    $007E,$0009,$3544,$0800
dc.w    $0000,$0004,$0000,$0008
dc.w    $0000,$0000,$13F0,$0000
dc.w    $007E,$000C,$F00E,$0000
dc.w    $0000,$0000,$0000,$0000
dc.w    $4000,$0000,$03F0,$0000
dc.w    $007E,$0002,$04C8,$0000
dc.w    $0080,$1000,$0000,$0000
dc.w    $0002,$0000,$03F0,$0000
dc.w    $007E,$0002,$4B00,$0000
dc.w    $0000,$0000,$0000,$0000
dc.w    $0000,$0020,$03F0,$0000
dc.w    $007E,$0008,$2028,$0000
dc.w    $0000,$4008,$2100,$0200
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$0007,$3A40,$0000
dc.w    $0000,$8800,$0000,$0000
dc.w    $0000,$0000,$43F0,$0000
dc.w    $007E,$0000,$0800,$0000
dc.w    $0000,$0000,$0800,$0000
dc.w    $0200,$8000,$03F0,$0000
dc.w    $007E,$0000,$0000,$0000
dc.w    $0000,$0000,$8000,$0000
dc.w    $0000,$0000,$03F0,$0000
dc.w    $007E,$0000,$0000,$0000
dc.w    $0000,$0000,$0000,$0040
dc.w    $0000,$0410,$03F0,$0000
dc.w    $007E,$0000,$0000,$0000
dc.w    $0000,$0000,$0008,$8000
dc.w    $1002,$0000,$03F0,$0000
dc.w    $007F,$FFFF,$FFFF,$FFFF
dc.w    $FFFF,$FFFF,$FFFF,$FFFF
dc.w    $FFFF,$FFFF,$FFF0,$0000
dc.w    $007F,$FFFF,$FFFF,$FFFF
dc.w    $FFFF,$FFFF,$FFFF,$FFFF
```

```
        dc.w    $FFFF,$FFFF,$FFF0,$0000
        dc.w    $007F,$FFFF,$FFFF,$FFFF
        dc.w    $FFFF,$FFFF,$FFFF,$FFFF
        dc.w    $FFFF,$FFFF,$FFF0,$0000
        dc.w    $007F,$FFFF,$FFFF,$FFFF
        dc.w    $FFFF,$FFFF,$FFFF,$FFFF
        dc.w    $FFFF,$FFFF,$FFF0,$0000
        dc.w    $003F,$FFFF,$FFFF,$FFFF
        dc.w    $FFFF,$FFFF,$FFFF,$FFFF
        dc.w    $FFFF,$FFFF,$FFE0,$0000
        dc.w    $001F,$FFFF,$FFFF,$FFFF
        dc.w    $FFFF,$FFFF,$FFFF,$FFFF
        dc.w    $FFFF,$FFFF,$FFC0,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000
        dc.w    $0000,$0000,$0000,$0000

o_k:    dc.l    text_ok,te,te
        dc.w    3,0,2,$11f0,0,3,5,0
```

```
text_ok:        dc.b   ' OK ',0
te:             dc.b   0


parent: dc.w    -1,1,2,20,0,16          ; large box
    dc.l        $00021100
    dc.w        100,26,340,80

    dc.w        2,-1,-1,23,0,0
    dc.l        b_map
    dc.w        25,6,260,90

    dc.w        0,-1,-1,22,7+32,0
    dc.l        o_k                     ; ok
    dc.w        200+20,35,70,14

aespb:  dc.l    contrl,global,intin,intout,addrin,addrout

object_draw:    dc.w   42,6,1,1,0
form_do:        dc.w   50,1,2,1,0

* GEM arrays
contrl:         ds.w   12
intin:          ds.w   128
intout:         ds.w   128
global:         ds.w   16
addrin:         ds.w   128
addrout:        ds.w   128

appl_init:      dc.w   10,0,1,0,0
appl_exit:      dc.w   19,0,1,0,0
form_center:    dc.w   54,0,5,1,0
```

Note that it is possible to include a resource file created by the RCP and hand constructed dialog boxes such as the one above. This allows the best of both worlds.

Chapter 17
GEM Windows

This chapter is devoted to GEM windows which most ST users have seen if they have ever used 1ST_WORD word processor or opened a window to see a disk drive directory. Many other applications use windows in one form or another.

Normally only four GEM windows can be opened at any one time but as desk accessories may also use one too, this results in a total of five. GEM provides the basis of window management: the components of a window and its many features such as scroll bars, and arrows etc. However, the programmer is left to deal with everything that goes on in and around the window: its re–drawing, updating and resizing, etc. If more than one window is open and they are overlapping the code to deal with this sort of situation is very complex as the contents of each window has to be refreshed or updated if any window is moved or resized.

The first GEM window example GEM15.S is shown below:

```
* GEM15.S
* This program opens a simple static GEM window
* Click close box to exit window.

* header
        move.l      a7,a5
        move.l      #ustk,a7
        move.l      4(a5),a5
        move.l      12(a5),d0
        add.l       20(a5),d0
        add.l       28(a5),d0
        add.l       #$100,d0
        move.l      d0,-(sp)
        move.l      a5,-(sp)
        clr.w       -(sp)
        move        #$4a,-(sp)
        trap        #1
        add.l       #12,sp
```

```
* appl_init()
        move.l      #appl_init,aespb
        jsr         aes     ; call AES

        jsr         mouse_off     ; turn mouse pointer off and
        jsr         arrow         ; change to arrow

* graf_handle()
        move.l      #graf_handle,aespb    ; get physical screen handle
        jsr         aes
        move        intout,gr_handle      ; store handle

* start by opening a virtual workstation
        move        #100,contrl
        move        #0,contrl+2
        move        #11,contrl+6

* is GDOS present
        moveq       #-2,d0
        trap        #2
        addq        #2,d0
        beq         no_gdos     ; no GDOS
        move        res,d0
        add         #2,d0
        move        d0,intin

        bra         s_no_gdos
no_gdos:

        move        #1,intin    ; default if GDOS not loaded

s_no_gdos:
        move        #1,intin+2    ; line type
        move        #1,intin+4    ; colour for line
        move        #1,intin+6    ; type of marking
        move        #1,intin+8    ; colour of marking
        move        #1,intin+10   ; character set
        move        #1,intin+12   ; text colour
        move        #1,intin+14   ; fill type
        move        #1,intin+16   ; fill pattern index
        move        #1,intin+18   ; fill colour
```

```
        move        #2,intin+20            ; coordinate flag
        move.w      gr_handle,contrl+12 ; device handle
        jsr         vdi                    ; v_opnvwk open virtual work station
        move.w      contrl+12,ws_handle ; store virtual workstation handle

* the type of the window
wtype           equ  $0ff

* the size lies in intout, so calculate the window size

* wind_get
        move.l      #wind_get,aespb
        move.w      #0,intin
        move        #5,intin+2     ; get window exterior coords
        jsr         aes

* wind_calc
        move        #1,intin               ; work position and size
        move.w      #wtype,intin+2
        movem.w     intout+2,d0-d3         ; returned from wind get
        movem.w     d0-d3,intin+4          ; the size
        move.l      #wind_calc,aespb
        jsr         aes

* now get its offsets
        move        intout+2,x
        move        intout+4,y
        move        intout+6,xwidth
        move        intout+8,ywidth

* and create the window
        move        #wtype,intin           ; see above
        movem       intout+2,d0-d3
        movem       d0-d3,intin+2          ; the size

* wind_create
        move.l      #wind_create,aespb
        jsr         aes
        move        intout,w_handle        ; save the handle
```

```
* now set its title
    move.w    w_handle,intin
    move.w    #2,intin+2          ; title string
    move.l    #windowname,intin+4 ; the address
    clr.w     intin+8
    clr.w     intin+10

* wind_set
    move.l    #wind_set,aespb
    jsr       aes

* set information title

    move.w    w_handle,intin
    move.w    #3,intin+2          ; information string

    move.l    #info,intin+4

    clr.w     intin+8
    clr.w     intin+10

* wind_set
    move.l    #wind_set,aespb
    jsr       aes

* now actually show it by opening it

    move.w    w_handle,intin
    movem.w   x,d0-d3
    add.w     #5,d0               ; x start
    movem.w   d0-d3,intin+2       ; the size

* wind_open
    move.l    #wind_open,aespb
    jsr       aes

* make interior of window white
* vsf_interior
    move      #23,contrl
    clr.w     contrl+2
    move.w    #1,contrl+6
```

```
        move.w      ws_handle,contrl+12
        move.w      #1,intin
        jsr         vdi

* vsf_stly
        move        #24,contrl
        clr.w       contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move.w      #1,intin
        jsr         vdi

* vsf_color
        move        #25,contrl
        clr.w       contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move.w      #0,intin
        jsr         vdi

*wind get first

        move.w      w_handle,intin
        move        #4,intin+2
        move.l      #wind_get,aespb
        jsr         aes
        movem.w     intout+2,d0-d3
        movem.w     d0-d3,x
        move.w      d0,ptsin
        move.w      d1,ptsin+2
        add.w       d2,d0
        add.w       d3,d1
        sub.w       #1,d0   ; adjust
        sub.w       #1,d1   ; adjust

* fill rect with white fill
        move.w      d0,ptsin+4
        move.w      d1,ptsin+6

* vr_recfl
        move        #114,contrl
```

```
        move.w   #2,contrl+2
        move.w   #0,contrl+6
        move.w   ws_handle,contrl+12
        jsr      vdi

        jsr      mouse_on

e_multi:
        move.l   #messagebuf,addrin
        move.l   #evnt_multi,aespb
        move     #1+2+16,intin  ; keyboard, mouse, report
        move     #1,intin+2     ; number of clicks
        move     #1,intin+4     ; left mouse button
        move     #1,intin+6     ; left button down
        move     #1,intin+8     ; leave rect (not applicable)

        move     #0,intin+10
        move     #0,intin+12
        move     #0,intin+14
        move     #0,intin+16
        move     #0,intin+18
        move     #0,intin+20
        move     #0,intin+22
        move     #0,intin+24
        move     #0,intin+26
        move     #0,intin+28
        move     #0,intin+30
        jsr      aes

        move.w   intout,d0      ; 2=mouse 1= k/b
        move.w   intout+2,mx    ; x mouse coord
        move.w   intout+4,my    ; y mouse coord
        cmpi.w   #$10,d0        ; mouse message
        beq      mouse
        cmpi.w   #2,d0          ; mouse button
        beq      e_multi

        move     intout+10,d1   ; key code
mouse:
        move.l   #messagebuf,a0
        move.w   (a0),d0
```

```
        cmpi.w      #$16,d0         ; L/Hand corner of window
        beq         quit
        bra         e_multi

* subroutines

vdi:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #vdipb,d1
        moveq.l     #$73,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

aes:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #aespb,d1
        move.w      #$c8,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

mouse_off:
        movem.l     a0-a4/d0-d5,-(sp)
        dc.w        $a000
        move.l      4(a0),a1
        move.l      8(a0),a2
        dc.w        $a00a
        movem.l     (sp)+,a0-a4/d0-d5
        rts

mouse_on:
        movem.l     a0-a4/d0-d5,-(sp)
        dc.w        $a000
        move.l      4(a0),a1
        move.l      8(a0),a2
        clr.w       (a2)
        clr.w       2(a1)
        clr.w       6(a1)
        dc.w        $a009
        movem.l     (sp)+,a0-a4/d0-d5
```

```
        rts

arrow:
        move.l      #graf_mouse,aespb
        move        #0,intin
        jsr         aes
        rts
* end of subroutines

quit:
* wind_close
        move.w      w_handle,intin
        move.l      #wind_close,aespb
        jsr         aes

* wind_delete
        move.w      w_handle,intin
        move.l      #wind_delete,aespb
        jsr         aes

* close the virtual workstation
* v_clsvwk
        move        #101,contrl
        clr.w       contrl+2
        clr.w       contrl+6
        move.w      ws_handle,contrl+12
        jsr         vdi

* appl_exit()
        move.l      #appl_exit,aespb
        bsr         aes     ; call AES

* now quit to the desktop
        clr.w       -(a7)
        trap        #1

        ds.l    100
ustk:   ds.l    1

* keep these dc.w together
x:      ds.w 1
```

```
y:              ds.w 1
xwidth:         ds.w 1
ywidth:         ds.w 1

w_handle:       ds.w 1
ws_handle:      ds.w 1
messagebuf:     ds.b 16

windowname:     dc.b    'Example Window',189,0

vdipb:          dc.l    contrl,intin,ptsin,intout,ptsout

contrl:         ds.w    128
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    128
addrin:         ds.w    128
addrout:        ds.w    128
ptsin:          ds.w    128
ptsout:         ds.w    128

aespb:          dc.l    contrl,global,intin,intout,addrin,addrout

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
evnt_multi:     dc.w    25,16,7,1,0
wind_get:       dc.w    104,2,5,0,0
wind_calc:      dc.w    108,6,5,0,0
wind_create:    dc.w    100,5,1,0,0
wind_set:       dc.w    105,6,1,0,0
wind_open:      dc.w    101,5,5,0,0
graf_handle:    dc.w    77,0,5,0,0
graf_mouse:     dc.w    78,1,1,1,0
wind_close:     dc.w    102,1,1,0,0
wind_delete:    dc.w    103,1,1,0,0

gr_handle:      ds.w    1
mx:             ds.w    1
my:             ds.w    1

info:           dc.b    ' Information area:',0
```

```
res:    ds.w    1
```

By studying the above source code you should be able to understand the process of opening a GEM window. As you can see it quite a laborious process, but in that process we are given much flexibility in the size and type of window that we can create.

The disk contains further information about the GEM calls made in the above program.

Note that the window must be filled with a colour, in this case white, otherwise the window appears with the background colour. This is why the 'vsf_interior' call and the other VDI calls are made.

There are a few new calls. The mouse_on and mouse_off routines in particular are very useful. Without switching the mouse off when drawing a window and its interior the mouse would be overdrawn but as soon as the mouse pointer was moved a gap would appear where the mouse originally was. These routines were found sometime ago in a Public Domain program and they work extremely well. They are somewhat better than the equivalent GEM calls as a count of how many times a mouse is hidden or shown has to be made in order to control the GEM mouse hide/show routines correctly.

'wind_calc' and 'wind_create' both use the equate:

```
wtype           equ  $0ff
```

'wtype' allows the type of window to be determined by the bits in its value, where a bit that is on stands for an active window component whilst a bit that is off is used for an inactive window component.

The bits have the following meaning:

Bit	value if on	meaning
0	1	title line with name of window
1	2	close box
2	4	full box
3	8	move box
4	16	information line

5	32	size box
6	64	up arrow
7	128	down arrow
8	256	vertical slider
9	512	left arrow
10	1024	right arrow
11	2048	horizontal slider

So if the value of 'wtype' was set to %00101111, or $2f then the title line, close box, full box, move box, and, size box will be drawn. All other components will be missing.

The message buffer 'messagebuf' the address of which is passed to the 'evnt__multi' routine holds all the messages that are passed to GEM when any actions are taken by the user with the mouse. See also chapter fifteen with shows its use with drop down menus.

The message received follows this format:

Element number	**Contents**
0	message id which indicates type of message
1	application id
2	number of additional bytes in excess of standard 16
3–7	depends on message

These are the events that are received in the message buffer:

Message number	**name**	**message**
10	mn__selected	menu item selected
20	wm__redraw	window display needs redrawing
21	wm__topped	a window has been selected to be the active, ie top, window
22	wm__closed	the close box has been clicked
23	wm__fulled	the full box has been clicked
24	wm__arrowed	the scroll bar or arrows have been clicked
25	wm__hslid	the horizontal scroll bar has been moved
26	wm__vslid	the vertical scroll bar has been moved
27	wm__sized	the bottom right size box has been

dragged

28	wm_moved	the move bar has been dragged
29	wm_newtop	a window has become active
40	ac_open	a desk accessory has been selected
41	ac_close	a desk accessory has been closed

The disk contains a tutorial on GEM by Tim Oren who was one of the original DR programmers that helped to write GEM. Initially it is heavy going for the beginner, especially as it refers to the C language. But this information is extremely useful and once you get used to the terminology much of the information can be used by the assembly language programmer.

The next piece of source code shows 'wm_sized' and the redrawing of the window.

First though clipping should be looked at.

Clipping

Clipping is a very useful concept and is used extensively in GEM window graphic operations. Say for instance we had a GEM window open that occupied the full screen and we wanted to draw a box using one of the VDI graphic primitives in the window. Now if the clipping function is set to the inner window's coordinates then when the box is drawn it can never go beyond the dimensions of the window. This is very useful for two good reasons. First we would not want a box drawn over the menu bar or scroll bars or beyond the dimensions of a particular window. it would be no good at all if part of the box was drawn in an entirely different window. Secondly, drawing anything beyond the dimensions of the screen is extremely dangerous as we do not know what occupies the memory there. It could be part of our program, and placing a box there would surely obliterate anything there with the resultant crash when we came to use that code!

If the window is moved or resized by the user it is a simple matter to set the clipping the dimensions of the window as the next example source code demonstrates.

```
* GEM16.S
* This program opens a simple GEM window and allows the user to
* resize it. A rectangle is drawn in the window, clipping is set
* so that the rectangle is only drawn in the window.
* Click close box to exit window.

* header
    move.l     a7,a5
    move.l     #ustk,a7
    move.l     4(a5),a5
    move.l     12(a5),d0
    add.l      20(a5),d0
    add.l      28(a5),d0
    add.l      #$100,d0
    move.l     d0,-(sp)
    move.l     a5,-(sp)
    clr.w      -(sp)
    move       #$4a,-(sp)
    trap       #1
    add.l      #12,sp

* appl_init()
    move.l     #appl_init,aespb
    jsr        aes      ; call AES
    move       intout,ap_id

    jsr        mouse_off      ; turn mouse pointer off and
    jsr        arrow          ; change to arrow

* graf_handle()
    move.l     #graf_handle,aespb  ; get physical screen handle
    jsr        aes
    move       intout,gr_handle      ; store handle

* start by opening a virtual workstation
    move       #100,contrl
    move       #0,contrl+2
    move       #11,contrl+6

* is GDOS present
    moveq      #-2,d0
```

```
    trap    #2
    addq    #2,d0
    beq     no_gdos   ; no GDOS
    move    res,d0
    add     #2,d0
    move    d0,intin

    bra     s_no_gdos
no_gdos:

    move    #1,intin  ; default if GDOS not loaded

s_no_gdos:
    move    #1,intin+2   ; line type
    move    #1,intin+4   ; colour for line
    move    #1,intin+6   ; type of marking
    move    #1,intin+8   ; colour of marking
    move    #1,intin+10  ; character set
    move    #1,intin+12  ; text colour
    move    #1,intin+14  ; fill type
    move    #1,intin+16  ; fill pattern index
    move    #1,intin+18  ; fill colour
    move    #2,intin+20  ; coordinate flag
    move.w  gr_handle,contrl+12 ; device handle
    jsr     vdi          ; v_opnvwk open virtual work station
    move.w  contrl+12,ws_handle ; store virtual workstation handle

* the type of the window
wtype       equ   $fff   ; all components
* the size lies in intout, so calculate the window size

* wind_get
    move.l  #wind_get,aespb
    move.w  #0,intin
    move    #5,intin+2
    jsr     aes

* wind_calc
    move    #1,intin
    move.w  #wtype,intin+2
```

```
        movem.w   intout+2,d0-d3         ; returned from wind get
        movem.w   d0-d3,intin+4          ; the size
        move.l    #wind_calc,aespb
        jsr       aes

* now get its offsets
        move      intout+2,x
        move      intout+4,y
        move      intout+6,xwidth
        move      intout+8,ywidth

* and create the window
        move      #wtype,intin           ; see above
        movem     intout+2,d0-d3
        movem     d0-d3,intin+2          ; the size

* wind_create
        move.l    #wind_create,aespb
        jsr       aes
        move      intout,w_handle        ; save the handle

* now set its title
        move.w    w_handle,intin
        move.w    #2,intin+2             ; title string
        move.l    #windowname,intin+4    ; the address
        clr.w     intin+8
        clr.w     intin+10

* wind_set
        move.l    #wind_set,aespb
        jsr       aes

        move.w    w_handle,intin
        move.w    #3,intin+2             ; information string

        move.l    #info,intin+4

        clr.w     intin+8
        clr.w     intin+10

* wind_set
```

```
        move.l    #wind_set,aespb
        jsr       aes

* now actually show it by opening it

        move.w    w_handle,intin
        movem.w   x,d0-d3
        add.w     #5,d0              ; x start
        movem.w   d0-d3,intin+2      ; the size

* wind_open
        move.l    #wind_open,aespb
        jsr       aes

* make interior of window white
        jsr       fill_window
        jsr       mouse_on

e_multi:
        move.l    #messagebuf,addrin
        move.l    #evnt_multi,aespb
        move      #1+2+16,intin ; keyboard, mouse, report
        move      #1,intin+2    ; number of clicks
        move      #1,intin+4    ; left mouse button
        move      #1,intin+6    ; left button down
        move      #1,intin+8    ; leave rect (not applicable)

        move      #0,intin+10
        move      #0,intin+12
        move      #0,intin+14
        move      #0,intin+16
        move      #0,intin+18
        move      #0,intin+20
        move      #0,intin+22
        move      #0,intin+24
        move      #0,intin+26
        move      #0,intin+28
        move      #0,intin+30
        jsr       aes

        move.w    intout,d0          ; 2=mouse 1= k/b
```

```
    move.w      intout+2,mx    ; x mouse coord
    move.w      intout+4,my    ; y mouse coord
    cmpi.w      #$10,d0        ; mouse message
    beq         mouse
    cmpi.w      #2,d0          ; mouse button
    beq         e_multi

    move        intout+10,d1   ; key code
mouse:
    move.l      #messagebuf,a0
    move.w      (a0),d0
    cmpi.w      #$16,d0        ; L/Hand corner of window
    beq         quit
    cmpi.w      #$1b,d0
    beq         resize
    bra         e_multi

resize:
* resize message received so get the new dimensions

    move        8(a0),intin+4
    move        10(a0),intin+6
    move        12(a0),intin+8
    move        14(a0),intin+10

    move        #5,intin+2

* wind_set
    move.w      w_handle,intin
    move.l      #wind_set,aespb
    jsr         aes

    jsr         fill_window

    jsr         draw_rounded_rect
    bra         e_multi

* the subroutines

draw_rounded_rect:
```

```
* set clip to inside rect
* vsf_color
* set fill colour
        move        #25,contrl
        move        #0,contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move        #1,intin
        jsr         vdi

* v_rfbox
* filled rounded rectangle
        move        #11,contrl
        move        #2,contrl+2
        move.w      #0,contrl+6
        move        #9,contrl+10            ; GDP function 9
        move.w      ws_handle,contrl+12
        move.w      #120,ptsin              ; x coord
        move        #110,ptsin+2            ; y coord
        move.w      #120+60,ptsin+4         ; x coord+width
        move        #110+80,ptsin+6         ; y coord+height
        jsr         vdi
        rts

fill_window:
* vs_clip
* set clip to inside rect
        move        #129,contrl
        move.w      #2,contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move        #0,intin    ; clipping off
        jsr         vdi

* vsf_interior
        move        #23,contrl
        clr.w       contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move.w      #1,intin
        jsr         vdi
```

```
* vsf_style
        move        #24,contrl
        clr.w       contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move.w      #1,intin
        jsr         vdi

* vsf_color- white
        move        #25,contrl
        clr.w       contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move.w      #0,intin
        jsr         vdi

* wind get first, get window internal dimensions
        move.w      w_handle,intin
        move        #4,intin+2
        move.l      #wind_get,aespb
        jsr         aes
        movem.w     intout+2,d0-d3
        movem.w     d0-d3,x
        move.w  d0,ptsin
        move.w      d1,ptsin+2
        add.w       d2,d0
        add.w       d3,d1
        sub.w       #1,d0
        sub.w       #1,d1

* fill rect with white fill
        move.w      d0,ptsin+4
        move.w      d1,ptsin+6
* vr_recfl
        move        #114,contrl
        move.w      #2,contrl+2
        move.w      #0,contrl+6
        move.w      ws_handle,contrl+12
        jsr         vdi

* vs_clip
```

```
* set clip to inside rect
        move        #129,contrl
        move.w      #2,contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move        #1,intin     ; clipping on
        jsr         vdi
        rts

vdi:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #vdipb,d1
        moveq.l     #$73,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

aes:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #aespb,d1
        move.w      #$c8,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

mouse_off:
        movem.l     a0-a4/d0-d5,-(sp)
        dc.w        $a000
        move.l      4(a0),a1
        move.l      8(a0),a2
        dc.w        $a00a
        movem.l     (sp)+,a0-a4/d0-d5
        rts

mouse_on:
        movem.l     a0-a4/d0-d5,-(sp)
        dc.w        $a000
        move.l      4(a0),a1
        move.l      8(a0),a2
        clr.w       (a2)
        clr.w       2(a1)
```

```
        clr.w       6(a1)
        dc.w        $a009
        movem.l     (sp)+,a0-a4/d0-d5
        rts

arrow:
        move.l      #graf_mouse,aespb
        move        #0,intin
        jsr         aes
        rts
* end of subroutines

quit:
* wind_close
        move.w      w_handle,intin
        move.l      #wind_close,aespb
        jsr         aes

* wind_delete
        move.w      w_handle,intin
        move.l      #wind_delete,aespb
        jsr         aes

* close the virtual workstation
* v_clsvwk
        move        #101,contrl
        clr.w       contrl+2
        clr.w       contrl+6
        move.w      ws_handle,contrl+12
        jsr         vdi

* appl_exit()
        move.l      #appl_exit,aespb
        bsr         aes         ; call AES

* now quit to the desktop
        clr.w       -(a7)
        trap        #1

                    ds.l   100
ustk:               ds.l      1
```

```
* keep these dc.w together
x:              ds.w    1
y:              ds.w    1
xwidth:         ds.w    1
ywidth:         ds.w    1

w_handle:       ds.w    1
ws_handle:      ds.w    1
messagebuf:     ds.b    16

windowname:     dc.b    'Example Window',', ',189,'somebody',0

vdipb:          dc.l    contrl,intin,ptsin,intout,ptsout

contrl:         ds.w    128
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    128
addrin:         ds.w    128
addrout:        ds.w    128
ptsin:          ds.w    128
ptsout:         ds.w    128

aespb:          dc.l    contrl,global,intin,intout,addrin,addrout

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
evnt_multi:     dc.w    25,16,7,1,0
wind_get:       dc.w    104,2,5,0,0
wind_calc:      dc.w    108,6,5,0,0
wind_create:    dc.w    100,5,1,0,0
wind_set:       dc.w    105,6,1,0,0
wind_open:      dc.w    101,5,5,0,0
graf_handle:    dc.w    77,0,5,0,0
graf_mouse:     dc.w    78,1,1,1,0
wind_close:     dc.w    102,1,1,0,0
wind_delete:    dc.w    103,1,1,0,0
appl_write:     dc.w    12,2,1,1,0
gr_handle:      ds.w    1

mx:             ds.w    1
```

```
my:     ds.w    1

info:   dc.b    ' Information area:',0
res:    ds.w    1
ap_id:  ds.w    1
```

In the above program two subroutines handle the interior fill of the window, and the drawing of a rectangle. They are 'fill__window:' and 'draw__rounded__rect:' respectively. Each time the window is resized the window must be filled with a white interior and then the rectangle drawn. To ensure the rectangle does not overwrite any part of the window clipping is set to the interior dimensions of the window. Note clipping is turned off when filling a newly resized window; if it was not then the fill would be bound by the dimensions of the previous interior.

The new dimensions of the window are found directly from a0 (from move.l #messagebuf,a0), and can be placed into the intin array by the method outlined below. Looking at the first line of code the offset 8 is added to the address held in a0, and then the contents of that address are placed in intin+4. Register a0 is not affected by this operation.

```
* resize message received so get the new dimensions

        move        8(a0),intin+4
        move        10(a0),intin+6
        move        12(a0),intin+8
        move        14(a0),intin+10
```

Next the 'wind__set' AES function is called to redraw the window at its new position.

```
        move        #5,intin+2
* wind_set
        move.w      w_handle,intin
        move.l      #wind_set,aespb
        jsr         aes
```

GDP's

The VDI supports ten basic drawing operations called **G**eneralized **D**rawing **P**rimitives. The function 'v__rfbox' is one of them. Others include filled circle, filled ellipse and justified text. Please see disk for a list of all the VDI library functions. The next chapter looks a the VDI in greater detail.

Chapter 18
Interfacing with GFA BASIC

This chapter looks at linking the object files produced by the zzSoft assembler with object files (see chapter 24 for more details about object files) produced by the GFA BASIC Version 3 compiler. Calling assembler routines from GFA BASIC V3 is also examined.

The zzSoft assembler produces DR (Digital Research) compatible object files which may be linked with similar GFA BASIC files, although there are a few constraints according to the GFA compiler: registers a3 to a6 must not be altered or used in the assembly language program.This leaves all the data registers though. However, registers a3–a6 may be saved to either an array or the stack (using the +$c GFA option) and be restored when the routine has finished.

Although the zzSoft assembler can be used to provide linkable DR compatible object files the purpose of the assembler is to provide a tool for the learning of assembler and is not guaranteed in any form or way if the reader uses it to produce object files for use with GFA BASIC programs.

The procedure for linking object files is outlined below using previous examples from earlier on in the book. Note that the object file to be linked must end in an 'rts' and begin with a global label that is called from GFA BASIC. Any assembly language file with a GEM header, stack, exit code eg 'pterm', any VDI initialisation, and 'appl_init' and 'appl_exit' must also be stripped out. GFA2.S shows this.

When using the AES and VDI the GEM arrays used by the program have also been initialized by GFA BASIC so should not really need reserving again but as there is no hooks into GFA regarding the GEM arrays it would appear better to duplicate them.

```
* GFA1.S
* This program finds the address of the screen, prints 'my_name'
* string to screen, clears the screen, and exits.
```

```
gemdos      equ   1
bios        equ   13
xbios       equ   14
cconws      equ   9
pterm       equ   $4c
con         equ   2

    .globl      start
start: move     #2,-(sp)            ; get screen RAM address
* returned in d0
    trap        #14                 ; call Xbios
    addq.l      #2,sp               ; correct stack
    move.l      d0,screen_address   ; put screen address in symbol

    move.l      #my_name,-(sp)      ; put address of string on stack
    move.w      #cconws,-(sp)       ; Gemdos function 'print a line'
    trap        #gemdos
    addq.l      #6,sp               ; correct stack

* this goes to the address 'wait_for_key_press' and executes the short
* routine held there until an 'rts' is found.
    jsr   wait_for_key_press

* lets clear the screen
    move.l      #31999,d0           ; counter #32000-1
    move.l      screen_address,a0   ; place screen address in an
* address register

do_it_again:
    clr.b       (a0)+               ; now clear the screen
    dbra        d0,do_it_again

* wait for a key press so that we can see the screen being cleared
* before returning to GFA BASIC.

    jsr         wait_for_key_press
    rts             ; this is here for GFA BASIC

************* subroutine *****************
wait_for_key_press:
* wait for key press subroutine
```

```
        move      #con,-(sp)      ; device number (console)
        move      #2,-(sp)        ; BIOS routine number
        trap      #bios           ; Call Bios
        addq.l    #4,sp
        rts

my_name:          dc.b   "Roger Pearson",0
screen_address:   ds.l   1
```

The assembly language routine should start with a label which must be declared as a global one by using the '.globl' directive. If this is not used then the object file will not contain any reference to such a label and the GFA BASIC program will not be able to find it. In effect the GFA BASIC compiler produces an object file of the GFA BASIC program and then links the above file with the GFA BASIC one and produces an executable file. Many files can be linked in this manner. The assembly language routine must contain an 'rts' at the end of the routine.

The GFA BASIC program

```
' clear screen
' GFA1.GFA
clear_screen
PROCEDURE clear_screen
  $X start
RETURN
```

To link the above files, first the assembly language file should be assembled and an object file produced (GFA1.O). Next the GFA BASIC compiler should be run and the above GFA program (GFA1.GFA) selected from the *File* drop down menu. 'C–Object C' should then be selected from the *Sets* menu and 'GFA1' (no need for .O extension) entered when prompted, and Return pressed to finish text entry. This file name should now be shown in the compiler options box top left 'Lnk: GFA1'. Next 'PRG=GFA F2' should be ticked so that the resultant '.PRG' file will take its name from the GFA file, and GFA1.PRG will be generated. Press F10 to complile GFA1.GFA and link GFA1.O which will produce GFA1.PRG.

One of the benefits of linking object files produced with the zzSoft assembler is that the 'dc' or 'ds' directives can be used in the source code, whereas when calling assembler from GFA BASIC (see later) they cannot be used as no relocation information can be given to GFA BASIC so that the labels can be used from their correct addresses. For instance the label 'my__name' refers to the address in memory that the string "Roger Pearson" is held at. Until the program is loaded the actual address is not calculated as the program can be loaded anywhere in the ST's free memory depending what desk accessories have been loaded etc. When the program is loaded (relocation) information is used to calculate the actual address of 'my__name'. The information is generated by the linker at assembly time, and appended to the '.PRG' file.

If a reference to a label is made from one object file to another the label in the second file must be declared global. Thus if the instruction 'jsr do__something' is made in object file one, then the label 'do__something' in the second file must be declared global by using the 'globl' directive, as in the above example with 'start:'

When producing linkable object files the source code hardly needs much adjustment, and many useful object file subroutines may be built up. Whereas when calling assembler from GFA BASIC great care has to used to ensure that no reference to addresses in memory that are unknown at assembly are used. This limits the usefulness of calling assembler from BASIC.

GFA2.S and GFA2.GFA shows example files that can be linked using the GFA BASIC compiler:

```
* GFA2.S
* This program displays a dialog box with a bit image in it.
* The dialog box and bit mapped image are both constructed from
* first principles.

    .globl      dialog
dialog:
* get res
    move        #4,-(sp)
    trap        #14
    addq.l      #2,sp
* res returned in d0
```

```
    cmp       #2,d0                ; is it high res
    bne       dont_alter_coords    ; no

    move.l    #parent,a0          ; address of tree in a5
    move.l    #9,d0               ; number of objects
    bsr       alter_coords

dont_alter_coords:
    bsr       f_center
    bsr       obdraw
    bsr       f_do
    rts

obdraw:
    move      #0,intin
    move      #2,intin+2
    move      d0,intin+4
    move      d1,intin+6
    move      d2,intin+8
    move      d3,intin+10
    move.l    #parent,addrin
    move.l    #object_draw,aespb
    bsr       aes
    rts

f_do: move.l  #form_do,aespb
    clr.w     intin        ; no editable text field
    move.l    #parent,addrin
    bsr       aes
    rts

f_center:
    move.l    #form_center,aespb
    move.l    #parent,addrin
    jsr       aes
    movem.w   intout+2,d0-d3
    rts

alter_coords:
    cmpi.b    #1,done_it
```

```
        beq         done
alter2_coords:
        move.b      #1,done_it
* adjust object data for high res screen
        add.l       #18,a0
        move.w      (a0),d1
        mulu.w      #2,d1
        move        d1,(a0)+
        add.l       #2,a0
        move        (a0),d1
        mulu.w      #2,d1
        move        d1,(a0)+
        dbf         d0,alter2_coords
done:
        rts

* AES subroutine

aes:    move.l      #aespb,d1
        move.l      #$c8,d0
        trap        #2
        rts

tl:     dc.l  t_1
        dc.w  4,16,0,0,$01f1

t_1:    dc.l  $00000000
        dc.l  $00000000
        dc.l  $ffffffff
        dc.l  $ffffffff
        dc.l  $03c00000
        dc.l  $03c0ff00
        dc.l  $03c1ff80
        dc.l  $03c1c380
        dc.l  $03c1c380
        dc.l  $03c1ff80
        dc.l  $03c1ff80
        dc.l  $03c1c780
        dc.l  $03c1c3c0
        dc.l  $03c1c1e0
        dc.l  $000000f0
```

```
        dc.l    $ffffffff

t2:     dc.l    t_2
        dc.w    4,16,0,0,$01f1

t_2:    dc.l    $00000000
        dc.l    $00000000
        dc.l    $ffffffff
        dc.l    $ffffffff
        dc.l    $00000000
        dc.l    $3fc1ff00
        dc.l    $7fe1ff80
        dc.l    $70e1c380
        dc.l    $70e1c380
        dc.l    $7fe1c380
        dc.l    $7fe1c380
        dc.l    $70e1c380
        dc.l    $70e1ff80
        dc.l    $70e1ff00
        dc.l    $00000000
        dc.l    $ffffffff

t3:     dc.l    t_3
        dc.w    4,16,0,0,$01f1

t_3:    dc.l    $00000000
        dc.l    $00000000
        dc.l    $ffffffff
        dc.l    $ffffffff
        dc.l    $00000000
        dc.l    $7fe1fe00
        dc.l    $7fe1ff80
        dc.l    $7001c380
        dc.l    $7001c380
        dc.l    $7fe1ff80
        dc.l    $7fe1ff80
        dc.l    $7001c780
        dc.l    $7fe1c3c0
        dc.l    $7fe1c1e0
        dc.l    $000000f0
        dc.l    $ffffffff
```

```
t4:     dc.l   t_4
        dc.w   4,16,0,0,$01f1

t_4:  dc.l   $00000000
      dc.l   $00000000
      dc.l   $ffffffff
      dc.l   $ffffffff
      dc.l   $00000000
      dc.l   $7fc1ffe0
      dc.l   $7fc1ffe0
      dc.l   $78001e00
      dc.l   $1e001e00
      dc.l   $03801e00
      dc.l   $03c01e00
      dc.l   $03c01e00
      dc.l   $3fc01e3c
      dc.l   $3fc01e3c
      dc.l   $00000000
      dc.l   $ffffffff

title1:            dc.b   ' Integrated Accounts',0

title2:            dc.b   ' Software',0

t6:     dc.l   ty,null,null
        dc.w   3,0,2,$13b2,0,1,14,0  ; gives red background, blue text

ty:     dc.b   'Version: 1.00',0

t7:     dc.b   189,' Someones Software',191,' 1990',0

null:   dc.b   0

exit_:  dc.l       text_ok,null,null
        dc.w       3,0,2,$1202,0,3,5,0

text_ok:           dc.b   'OK',32,175,0

aespb:  dc.l   contrl,global,intin,intout,addrin,addrout

object_draw:       dc.w   42,6,1,1,0
```

```
form_do:        dc.w    50,1,2,1,0

parent:         dc.w    -1,1,9,20,0,16        ; large box
    dc.l        $22020
    dc.w        170,50,250,120

    dc.w        2,-1,-1,28,0,0
    dc.l        title1
    dc.w        35,30,90,15

    dc.w        3,-1,-1,23,0,0        ; 23=bitblk
    dc.l        t1
    dc.w        10,10,16,19

    dc.w        4,-1,-1,23,0,0
    dc.l        t2
    dc.w        40,10,16,19

    dc.w        5,-1,-1,23,0,0
    dc.l        t3
    dc.w        70,10,16,19

    dc.w        6,-1,-1,23,0,0
    dc.l        t4
    dc.w        100,10,16,19

    dc.w        7,-1,-1,28,0,0
    dc.l        title2
    dc.w        70,40,90,15

    dc.w        8,-1,-1,22,7,0
    dc.l        exit_             ; exit
    dc.w        100,100,50,15

    dc.w        9,-1,-1,22,0,0
    dc.l        t6                ; version
    dc.w        50,60,150,15

    dc.w        0,-1,-1,28,32,0
    dc.l        t7                ; (c) copyright
    dc.w        30,80,150,15
```

*** GEM arrays**

```
contrl:       ds.w  12
intin:        ds.w  128
intout:       ds.w  128
global:       ds.w  16
addrin:       ds.w  128
addrout:      ds.w  128

form_center:  dc.w  54,0,5,1,0
done_it:      ds.b  1
```

Note no GEM header is used and the routine ends in a 'rts'. Also, each time the routine is called the dialog tree would be adjusted for high res (if hi res was used) so a flag is used to deflect the course of the program so that it does not do it again when the routine is called again. The symbol 'done__it' is used. A 'flag' is a term used to mean that some state is either on or off, rather like a 'go' or 'stop' signal is used to tell a train driver or motorists to either proceed or halt. By testing to see if 'done__it' contains a one or not the program can be controlled to our wishes. Register a5 in the coords adjust routine has been altered to register a0 to conform with the requirements of GFA BASIC.

The GFA program:

```
' GFA2.GFA
' display dialog box
get_dialog
PROCEDURE get_dialog
  $X dialog
RETURN
```

The rest of the chapter looks at calling assembly language routines from assembler by including the '.PRG' files as data statements in the GFA BASIC source code.

The method to do this is:

1. Assemble the source code to produce a '.PRG' file.

2. Turn this into data statements using the GFA BASIC utility to do this.

3. Run the program in the interpreter or compile and run.

For instance examine the assembly language program below which calls another executable file using the 'p_exec' call and executes it immediately. The source code for both programs are listed.

```
* P_EXEC0.S
* This program loads and executes another

* header
    move.l      a7,a5
    move.l      #ustk,a7
    move.l      4(a5),a5
    move.l      12(a5),d0
    add.l       20(a5),d0
    add.l       28(a5),d0
    add.l       #$100,d0
    move.l      d0,-(sp)
    move.l      a5,-(sp)
    clr.w       -(sp)
    move        #$4a,-(sp)
    trap        #1
    add.l       #12,sp

* p_exec  start and run
    move.l      #env,-(sp)
    move.l      #com,-(sp)
    move.l      #fil,-(sp)
    move        #0,-(sp)       ; load and run immediately
    move        #$4b,-(sp)
    trap        #1
    add.l       #16,sp

* quit
    move        #7,-(sp)
    move        #$4c,-(sp)
    trap        #1
```

```
        ds.l    20
ustk:   ds.l    1

        .globl       env
env:    dc.b         0
        .globl       com
com:    dc.b         0
        .globl       fil
fil:    dc.b         '2.prg',0
```

The program 2.prg source code:

```
* 2.s
* simple program
* display 'A' on screen

    move      #65,-(sp)
    move      #2,-(sp)
    trap      #1
    addq.l    #4,sp

*wait for key press
    move      #1,-(sp)
    trap      #1
    addq.l    #2,sp

* exit pterm
    move      #3,-(sp)   ; exit code
    move      #$4c,-(sp)
    trap      #1
    addq.l    #4,sp
```

So what happens is that the first program loads the second, '2.prg' and it is immediately executed, and an 'A' is displayed on screen, and when a key press is processed the program exits back to the calling program with the exit code three and then the calling program exits.

To use the 'p__exec0' program from GFA BASIC V3 there is a number of methods we can use:

The first method is 'C:addr([x,y,...])

where the function 'C:' calls an assembler subroutine located at address 'addr'. Parameters may be passed via the stack, as either longs or words. The first long passed to via the stack is the return address. See GFA BASIC book for more details.

```
' EXEC_0A.GFA
' using assembly language from GFA BASIC EXEC_0A.PRG
' passing paramters USING C:ADDR)[X,Y])
'
DIM asm%(68/2)
asm_adr%=V:asm%(0)
adr%=asm_adr%
DO
  READ asm%
  EXIT IF asm%=-1
  CARD adr% =asm%
  ADD adr%,2
LOOP
'
'
x$=""
x%=V:x$
z$="2.prg"
z%=V:z$
exec:
~C:asm_adr%(L:x%,L:x%,L:z%)        ! data lengths should be specified in
' this case longs, (L:)
'
DATA 8303,4,8815,8,9327,12,12040,12041
DATA 12042,16188,0,16188,75,20033,57340,0
DATA 16,20085
DATA -1
```

To use the file for inclusion into a GFA BASIC program we have to prepare the assembly language source code before it is assembled:

```
* EXEC_0A.S
* passing parameters via the stack in GFA BASIC
* C:addr([x,y,...])
```

```
* addr: avar (at least 32 bit, ideally intger-type: adr%)
* x,y: iexp
* method #1
    move.l      4(sp),a0     ; get the parameters
    move.l      8(sp),a1
    move.l      12(sp),a2
* run
    move.l      a0,-(sp)     ; env
    move.l      a1,-(sp)     ; command line
    move.l      a2,-(sp)     ; file name
    move        #0,-(sp)     ; mode (load and run=0)
    move        #$4b,-(sp)
    trap        #1
    add.l       #16,sp       ; correct stack
    rts                      ; need this for GFA
```

This should be assembled using the zzSoft assembler and the resultant '.PRG' should be converted into data statements as shown above in EXEC__0A.GFA. A utility to do this is provided on the GFA BASIC disk – ASM__DATA.LST.

The second method is to use RCALL:

This is very useful as it allows the BASIC programmer to specify what values will be in any data or address register at the start of the assembler routine. When the routine returns it is also possible to inspect the same registers.

This is done by declaring an array of 16 longs called 'reg%. The data registers d0–d7 are then allocated to 'reg%(0)' to 'reg%(7)' and the address registers a0–a6 to 'reg%(8)' to reg%(14). See GFA BASIC V3 User Guide for more details.

The GFA BASIC program to do the same as the above two programs, P__EXEC0.S and EXEC__0A.GFA using the GFA function 'RCALL' is:

```
' EXEC_0B.GFA
' using assembly language from GFA BASIC EXEC_0B.PRG
' passing parameters USING RCALL ADDR,REG%()
```

```
'
DIM reg%(16)
DIM asm%(56/2)
asm_adr%=V:asm%(0)
adr%=asm_adr%
DO
  READ asm%
  EXIT IF asm%=-1
  CARD adr% =asm%
  ADD adr%,2
LOOP
'
x$=""
reg%(8)=V:x$   !A0
reg%(9)=V:x$   !A1
z$="2.prg"
reg%(10)=V:z$  !A2
exec:
RCALL asm_adr%,reg%()
'
' EXAMINE D0;
PRINT reg%(0)          ! 3 should be returned here
'
DATA 12040,12041,12042,16188,0,16188,75,20033
DATA 57340,0,16,20085
DATA -1
'
```

The assembly language source code to assemble and turn into data statements is:

```
* EXEC_0B.S
* passing parameters via the stack in GFA BASIC
* RCALL addr,reg%()
* reg%(): Name of integer (4-byte) array
* addr: iexp

* run
    move.l    a0,-(sp)     ; env
    move.l    a1,-(sp)     ; command line
    move.l    a2,-(sp)     ; file name
    move      #0,-(sp)     ; mode (load and run=0)
```

```
    move        #$4b,-(sp)
    trap        #1
    add.l       #16,sp      ; correct stack
    rts                     ; need this for GFA
```

Using labels that cannot be relocatable in assembly language programs to be included as data statements is not advisable but this does not include 'bsr', 'bra' and 'dbra' to a label which can be safely used. This allows the use of looping with an assembly language routine. Note that jumps to labels cannot be used eg, 'jsr' you should use 'bsr' instead.

Please examine the following source code as it is an example of assembly language source code that would not work with GFA BASIC as the addresses of the labels with 'dc.b' could not be known by GFA, for the reasons stated earlier.

```
* EXEC_0.S
* Assembly language source code for 'p_exec' mode 0, start and run
* Would not work with GFA BASIC
    move.l      #env,-(sp)      ; environment
    move.l      #com,-(sp)      ; command line
    move.l      a0,-(sp)        ; file name
    move        #0,-(sp)        ; mode
    move        #$4b,-(sp)
    trap        #1
    add.l       #16,sp          ; correct stack: note this is 16
    rts

env:            dc.b    0
com:            dc.b    0
* fil    dc.b    '2.prg',0    ; this would be passed via a0 by GFA
* BASIC call 'C:'
```

It is always advisable to save all registers to the stack before calling an assembly language routine from GFA BASIC. They can be retrieved at the end of the routine. This is easily done with the 'movem.l d0-d7/a0-6,-(sp)' instruction at the start of a routine, and the corresponding 'movem.l (sp)+, d0-d7/a0-a6' at the finish. See the rest of this book for many examples.

Chapter 19
The VDI

This chapter looks at using the VDI to output simple text, and a rounded rectangle to the screen. Blitting is also examined. Later chapters look at VDI output to printers. The reader is referred to chapter eleven for some details of the VDI.

```
* VDI1.S
* This program outputs the letter 'A' and a rectangle to the screen

*HEADER
    move.l      a7,a5
    move.l      #ustk,a7
    move.l      4(a5),a5
    move.l      12(a5),d0
    add.l       20(a5),d0
    add.l       28(a5),d0
    add.l       #$100,d0
    move.l      d0,-(sp)
    move.l      a5,-(sp)
    clr.w       -(sp)
    move        #$4a,-(sp)
    trap        #1
    add.l       #12,sp

* get current screen resolution
    move.w      #4,-(sp)
    trap        #14
    addq.l      #2,sp
    move.w      d0,res

* is gdos present
    moveq       #-2,d0
    trap        #2
    addq        #2,d0
    beq         no_gdos
    move        res,d0
```

```
        add         #2,d0
        move        d0,intin
no_gdos:


* graf_handle
        move.l      #graf_handle,aespb
        jsr         aes
        move        intout,gr_handle

* start by opening a virtual  workstation
        move        #100,contrl
        move        #0,contrl+2
        move        #11,contrl+6
        move        gr_handle,contrl+12
        move        #1,intin+2
        move        #1,intin+4
        move        #1,intin+6
        move        #1,intin+8
        move        #1,intin+10
        move        #1,intin+12
        move        #1,intin+14
        move        #1,intin+16
        move        #1,intin+18
        move        #2,intin+20
        jsr         vdi
        move.w      contrl+12,ws_handle

* appl_intit()
        move.l      #appl_init,aespb
        jsr         aes         ; call AES

* vst_point
* font height in points
        move        #107,contrl
        move.w      #0,contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move        #24,intin          ; height in points
        jsr         vdi
```

```
* v_gtext
* text output
    move      #8,contrl
    move      #1,contrl+2
    move      #1,contrl+6    ; number of chars in string
    move      ws_handle,contrl+12
    move      #20,ptsin      ; x coord screen
    move      #50,ptsin+2    ; y coord screen
    move      #65,intin      ; actual character='A'
    jsr       vdi

* v_rbox
* rounded rectangle
    move      #11,contrl
    move      #2,contrl+2
    move      #0,contrl+6
    move      ws_handle,contrl+12
    move      #8,contrl+10   ; function 8
    move      #100,ptsin     ; x coord screen
    move      #50,ptsin+2    ; y coord screen
    move      #100+60,ptsin+4 ; x coord right edge
    move      #50+40,ptsin+6  ; y coord bottom edge
    jsr       vdi

* wait for keypress(no echo)
    move      #8,-(sp)
    trap      #1
    addq.l    #2,sp

*  close the virtual workstation
    move      #101,contrl
    clr.w     contrl+2
    clr.w     contrl+6
    move.w    ws_handle,contrl+12
    jsr       vdi

* appl_exit()
    move.l    #appl_exit,aespb
    bsr       aes    ; call AES

quit:
```

```
        move        #1,-(sp)
        move        #$4c,-(sp)
        trap        #1

        ds.l    100
ustk:   ds.l    1

contrl:         ds.w    128
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    128
addrin:         ds.w    128
addrout:        ds.w    128
ptsin:          ds.w    128
ptsout:         ds.w    128

aespb:  dc.l    contrl,global,intin,intout,addrin,addrout
vdipb:  dc.l    contrl,intin,ptsin,intout,ptsout

aes:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #aespb,d1
        move.w      #$c8,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts
vdi:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #vdipb,d1
        moveq.l     #$73,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

p_handle:       ds.w    1
gr_handle:      ds.w    1
ws_handle:      ds.w    1
res:            ds.w    1
appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
graf_handle:    dc.w    77,0,5,0,0
```

This simple program uses two VDI calls to output to the screen, 'v_gtext', and 'v_rbox', the first graphic text, and the second a rounded rectangle. The height of the text can be set from the VDI call 'vst_point', which sets the height in points where a point is equal to 1/72". So 36 points is equal to 1/2". In this case the VDI uses the inbuilt ROM screen font. Later we will use a font loaded from disk.

The 'v_gtext' call is interesting as it lets the programmer place text anywhere on the screen. Text can even be placed beyond or before (use a negative value in 'ptsin+2') it. Obviously in cases like that it is sensible to ensure that clipping is set to the screen boundaries. Note that 'contrl+6' expects the number of characters in the string to be passed here. The actual string should be placed in the 'intin' array, as a word with actual ASCII character in the lower byte and with a null in the higher byte of the word. Any character with an image in the font (character set) can be sent to the screen as control characters are not recognized. This differs from the GEMDOS 'bconout' as passing the ASCII code 10 would result in the cursor being moved down one line – a line feed. This is because the ST emulates a DEC VT52 display terminal and therefore interprets any ASCII character from 0 to 31 as a non–printing control character. The VDI graphic text output do not follow this emulation and therefore the line feed character would be printed on the screen as a small bell. Note that a GEM program switches off the cursor so that it has to be turned on by using an escape code, so called because the display under VT52 emulation responds to strings of characters beginning with the character 27 (Esc). The escape for turning on the cursor is "e". So to turn the cursor on the code would be:

```
* cursor on
    move      #27,-(sp) ; escape
    move      #2,-(sp)  ; console
    move      #3,-(sp)  ; opcode- function number
    trap      #13,-(sp)
    addq.l    #6,sp

    move      #101,-(sp) ; 'e'
    move      #2,-(sp)  ; console
    move      #3,-(sp)  ; opcode- function number
    trap      #13,-(sp)
```

```
    addq.l    #6,sp
* cursor off
    move      #27,-(sp) ; escape
    move      #2,-(sp)  ; console
    move      #3,-(sp)  ; opcode- function number
    trap      #13,-(sp)
    addq.l    #6,sp

    move      #102,-(sp) ; 'f'
    move      #2,-(sp)  ; console
    move      #3,-(sp)  ; opcode- function number
    trap      #13,-(sp)
    addq.l    #6,sp
```

To place a string in the intin array it is possible to use 'evnt_multi', or 'evnt_keybd' to get the characters from the user and then place them in the intin array. Please see next example:

```
* VDI2.S
* This program outputs a VDI graphics string to the console

*HEADER
    move.l    a7,a5
    move.l    #ustk,a7
    move.l    4(a5),a5
    move.l    12(a5),d0
    add.l     20(a5),d0
    add.l     28(a5),d0
    add.l     #$100,d0
    move.l    d0,-(sp)
    move.l    a5,-(sp)
    clr.w     -(sp)
    move      #$4a,-(sp)
    trap      #1
    add.l     #12,sp

* appl_intit()
    move.l    #appl_init,aespb
    jsr       aes       ; call AES
```

```
* get current screen resolution
      move.w     #4,-(sp)
      trap       #14
      addq.l     #2,sp
      move.w     d0,res

* is gdos present
      moveq      #-2,d0
      trap       #2
      addq       #2,d0
      beq        no_gdos
      move       res,d0
      add        #2,d0
      move       d0,intin
no_gdos:

* graf_handle
      move.l     #graf_handle,aespb
      jsr        aes
      move       intout,gr_handle

* start by opening a virtual  workstation
      move       #100,contrl
      move       #0,contrl+2
      move       #11,contrl+6
      move       gr_handle,contrl+12
      move       #1,intin+2
      move       #1,intin+4
      move       #1,intin+6
      move       #1,intin+8
      move       #1,intin+10
      move       #1,intin+12
      move       #1,intin+14
      move       #1,intin+16
      move       #1,intin+18
      move       #2,intin+20
      jsr        vdi
      move.w     contrl+12,ws_handle

* vst_point
* font height in points
```

```
        move      #107,contrl
        move.w    #0,contrl+2
        move.w    #1,contrl+6
        move.w    ws_handle,contrl+12
        move      #18,intin      ; height in points
        jsr       vdi

* alert
        move.l    #form_alert,aespb
        move      #1,intin        ; first button
        move.l    #alert_string,addrin
        bsr       aes

        move.l    #intin,a0
* evnt_keybd
        clr.l     d3
again:
        move.l    #evnt_keybd,aespb
        jsr       aes         ; call AES
        move      intout,d0
        and       #$0ff,d0
        move      d0,(a0)+
        add       #1,d3       ; count the number of chars
        cmpi.w    #5,d3       ; allow user to enter 5 characters
        bne       again

* v_gtext
* output string in a0
        move      #8,contrl
        move      #1,contrl+2
        move      d3,contrl+6     ; number of chars in string
        move      ws_handle,contrl+12
        move      #120,ptsin      ; x coord screen
        move      #150,ptsin+2    ; y coord screen
        jsr       vdi

* wait for keypress(no echo)
        move      #8,-(sp)
        trap      #1
        addq.l    #2,sp
```

```
* close the virtual workstation
        move     #101,contrl
        clr.w    contrl+2
        clr.w    contrl+6
        move.w   ws_handle,contrl+12
        jsr      vdi

* appl_exit()
        move.l   #appl_exit,aespb
        bsr      aes          ; call AES

quit:
        move     #1,-(sp)
        move     #$4c,-(sp)
        trap     #1

        ds.l  100
ustk:   ds.l  1

contrl:          ds.w   128
intin:           ds.w   128
intout:          ds.w   128
global:          ds.w   128
addrin:          ds.w   128
addrout:         ds.w   128
ptsin:           ds.w   128
ptsout:          ds.w   128

aespb:  dc.l  contrl,global,intin,intout,addrin,addrout
vdipb:  dc.l  contrl,intin,ptsin,intout,ptsout

aes:
        movem.l  d0-d7/a0-a6,-(sp)
        move.l   #aespb,d1
        move.w   #$c8,d0
        trap     #2
        movem.l  (sp)+,d0-d7/a0-a6
        rts
vdi:
        movem.l  d0-d7/a0-a6,-(sp)
        move.l   #vdipb,d1
```

```
        moveq.l   #$73,d0
        trap      #2
        movem.l   (sp)+,d0-d7/a0-a6
        rts

p_handle:       ds.w   1
gr_handle:      ds.w   1
ws_handle:      ds.w   1
res:            ds.w   1
appl_init:      dc.w   10,0,1,0,0
appl_exit:      dc.w   19,0,1,0,0
evnt_keybd:     dc.w   20,0,1,0,0
graf_handle:    dc.w   77,0,5,0,0
form_alert:     dc.w   52,1,1,1,0

alert_string:   dc.b   "[3] Please type five letters |and then press any|"
                dc.b   "key to exit. |Click Ok first though! ][ Ok ]",0
```

In the above program the user types five characters at the keyboard via the 'evnt_keybd' call, which are then displayed by the 'v_gtext' VDI function.

To effect this procedure first the address of the intin array is first placed in a0 before the 'evnt_keybd' call. Register d3 is cleared as it is to used as a counter to hold the number of keypresses. The VDI function 'evnt_keybd' waits for a keypress and then passes the result via the intout array. The value passed by intout contains the ASCII character in the lower byte and the scan code in the higher byte of the low word – similar to 'conin' the GEMDOS call. The high byte contains a unique identifier of the key struck which is independent of whether the Shift, Control or Alt key was pressed whilst the lower byte contains the ASCII value, which does take into account whether the Shift key was pressed. However, we are only interested in the contents of the lower byte of intout as 'v_gtext' expects to receive an ASCII character bound as a word. The ASCII character should be placed in the low part of the word, whilst the high byte should contain a null (0) – not the ASCII character '0' which is different. The program fragment that does this is shown overpage:

```
     move.l     #intin,a0
* evnt_keybd
     clr.l      d3
again:
     move.l     #evnt_keybd,aespb
     jsr        aes       ; call AES
     move       intout,d0
     and        #$0ff,d0
     move       d0,(a0)+
     add        #1,d3     ; count the number of chars
     cmpi.w     #5,d3     ; allow user to enter 5 characters
     bne        again
```

What we need to do is to ensure that the intin array only receives the correct parameters. To do this we take the intout output and place it in register d0 and 'and' the contents of d0 with #$0ff. This is known as masking as 'anding' something masks off the part we require without doing anything to it and removes the part we don't want by changing all the bits to zero. The result of 'anding' register d0 with #$0ff is to cause register d0 to contain only the the ASCII character in the lower byte and nothing to be present in the higher byte of the low word. Note here we are only concerned with the lower word. It is irrelevant to us as to what the higher word may contain. Next the result of this operation is placed into the intin array – 'move d0,(a0)+' and the address of the intin array is incremented by a word ready for the next ASCII code to be placed. Then a one is added to the contents of register d3 and this is then checked to see if the maximum number of characters allowed has been achieved.

Logical and

To understand the 'and' operation we need to look at what we are doing here, which is in effect a Boolean operation on each bit of the lower word of d0 – a logical 'and'. A detailed description of Boolean arithmetic is beyond the scope of this book, but we can look at results of 'anding' which can be very useful in many assembly language operations.

Whenever a '1' is anded with a bit it always returns whatever is contained in that bit. So 'and #$01,d0' will return in d0 whatever is in bit 0 of d0. Whenever a '0' is anded with a bit it returns a '0'. In effect 'and #$01,d0' is the same as 'and #%00000001,d0' as the 'and' source can only contain zeros. This results in only the value of bit 0 being left or returned in d0.

So if we want to get the lower byte of a word in a register we just need to 'and' it with #$ff, which will return everything that is contained in the lower byte whilst zeroing everything else. This is also useful if we want to isolate a particular bit in a register, all we need to do is 'and' it with the relevant amount. So to return the value contained in bit 3 of register d0, we would 'and #$4,d0' or 'and #%00000100,d0'. Please note the zzSoft assembler can only accept the hex value.

Logical or

A logical 'or' is often used in a similar manner to a logical 'and' except this time instead of returning the original bits it is used to set bits.

So 'or #$01,d0' would ensure that bit 0 was set no matter what is was before. Note other instructions can be used such as 'bset' to set a bit, 'btst' to test a bit. See chapter eight which uses these instructions extensively in its example source code.

The VDI consists of many useful routines that can be utilised by the programmer: lines of various thicknesses, boxes, circles, etc. Please see disk for list of VDI calls.

Bit blitting

The VDI contains very useful functions that allows the programmer to manipulate bit image blocks, the bit blit operation (Bit BLock Transfer). The one we will be examining is called 'vro_cpyfm' – copy raster. This is very useful when moving parts of the screen about, for example in GEM windows when blocks of text need to be shifted about when scrolling. For instance a word processor like First Word displays its text in a GEM window. If the user had loaded a large text file into the window and was displaying only part of the file then the user may decide to scroll downward using the cursor at the bottom of the window. The block of text from the bottom of the window to the line before the first line would be moved (or blitted) so that the block of text would be moved up a line all at once. The bottom line – the next line the user wants to read would be displayed by other methods. This way very smooth scrolling can be achieved. Load First Word and have a look. The zzSoft text editor uses this technique for smooth scrolling.

Naturally art programs can make good use of these blitting techniques as can animation, and other programs. Note that the STE has a chip that speeds up blitting built into the hardware.

These functions called VDI raster operations by Atari, use a format called the Memory Form Definition Block (MFDB) to manipulate the data presented by the programmer. It is laid out in the following manner:

Long	address of image data
word	image width in pixels
word	image height in lines
word	image width in words
word	image format flag 0=ST specific format, 1=GEM format
word	number of colour bit planes
word	reserved for future use
word	reserved for future use
word	reserved for future use

The first member of the MFDB structure should contain the address of the bit image area. If the value here is zero the VDI recognizes this as then being referred to the screen and the rest of the array is filled in by the VDI.

The image width and height should be even multiples of 16 pixels wide. The image width in words is found by dividing the pixel value by 16 (16 bits = 1 word).

The next member of the structure shows whether image data is arranged in the format of the ST display memory, or in standard GEM format. 0= ST format, 1= GEM format.

The next member is the number of bit=planes used by the image. In high res this is 1, whilst in med res this is 2, and in low res this 4. The 'vq_extend' call can be used to determine the number of planes being used by the program by accessing the 'work_out' array. Please see the supplied disk. In the example source BLIT.S the resolution of the screen is first found and then the MFDB block is adjusted accordingly.

The VDI call looks like this in assembly language:

```
* vro_cpyfm
    move      #109,contrl
    move.w    #4,contrl+2
    move.w    #1,contrl+6
    move.w    ws_handle,contrl+12
    move.l    #mfdb1,contrl+14      ; source address
    move.l    #mfdb2,contrl+18      ; destination address
    move      #0,intin              ; xor mode

* source block coordinates using screen screen
    move      #0,ptsin        ; x coordinate
    move      #0,ptsin+2      ; y coordinate
    move      #640,ptsin+4    ; width
    move      ht,ptsin+6      ; height

* destination coordinates
    move      #0,ptsin+8
    move      #0,ptsin+10
    move      #640,ptsin+12
    move      ht,ptsin+14
    jsr       vdi
```

The MFDB data structure looks like this:

```
* structure to store a bit image DEGAS picture (high res)

mfdb1: dc.l    store_area    ; DEGAS pic
       dc.w    640           ; width in pixels
width: dc.w    400           ; height in pixles (high res)
       dc.w    40            ; divide pixel width by 16
       dc.w    0
res:   dc.w    2             ; number of planes
       dc.w    0
       dc.w    0
       dc.w    0
```

```
store_area:    ds.b 32000

mfdb2:         dc.l  screen_area
        dc.w   640
width:  dc.w   400
        dc.w   40
        dc.w   0
res:    dc.w   2
        dc.w   0
        dc.w   0
        dc.w   0
```

The following source code load a DEGAS picture from disk, stores it in a buffer, waits for a key press and then the picture is displayed on screen by blitting it from the DEGAS buffer.

```
* BLIT.S
* This program load a DEGAS file into a buffer and blits it from
* that buffer to the screen.

* HEADER
    move.l    a7,a5
    move.l    #ustk,a7
    move.l    4(a5),a5
    move.l    12(a5),d0
    add.l     20(a5),d0
    add.l     28(a5),d0
    add.l     #$100,d0
    move.l    d0,-(sp)
    move.l    a5,-(sp)
    clr.w     -(sp)
    move      #$4a,-(sp)
    trap      #1
    add.l     #12,sp

* get current screen resolution
    move.w    #4,-(sp)
    trap      #14
    addq.l    #2,sp
    move.w    d0,screen_res
```

```
* is gdos present
    moveq     #-2,d0
    trap      #2
    addq      #2,d0
    beq       no_gdos
    move      screen_res,d0
    add       #2,d0
    move      d0,intin
no_gdos:

* graf_handle
    move.l    #graf_handle,aespb
    jsr       aes
    move      intout,gr_handle

* start by opening a virtual workstation
    move      #100,contrl
    move      #0,contrl+2
    move      #11,contrl+6
    move      gr_handle,contrl+12
    move      #1,intin+2
    move      #1,intin+4
    move      #1,intin+6
    move      #1,intin+8
    move      #1,intin+10
    move      #1,intin+12
    move      #1,intin+14
    move      #1,intin+16
    move      #1,intin+18
    move      #2,intin+20
    jsr       vdi
    move.w    contrl+12,ws_handle

* appl_init()
    move.l    #appl_init,aespb
    jsr       aes          ; call AES

    move      #400,ht

    cmp       #2,screen_res          ; is it high res
    beq       dont_alter_coords      ; no
```

```
* alter MFDB values
     move        #200,ht         ; med res screen height
     move        #200,width
     move        #200,wdth
     move        #2,res
     move        #2,res_

dont_alter_coords:
*open
     move.w      #0,-(sp)            ; set file attribute
     move.l      #file_name,-(sp)    ; address of filename
     move.w      #$3d,-(sp)          ; open function number
     trap        #1           ; hello GEMDOS
     add.l       #8,sp
     tst         d0           ; -ve number?
     bmi         general_error       ; yes, go to error routine
     move.w      d0,handle           ; store file handle

* read palette data
     move.l      #pic_header,-(sp)   ; pic_header address
     move.l      #34,-(sp)           ; number of bytes to read
     move        handle,-(sp)
     move.w      #$3f,-(sp)
     trap        #1
     add.l       #12,sp
     tst         d0                  ; -ve number?
     bmi         general_error       ; yes, go to error routine

* use new palette
     move.l      #pic_header+2,-(sp)     ; address of palette
     move        #6,-(sp)            ; set palette
     trap        #14                 ; call Xbios
     add.l    #6,sp

*  get screen address
     move        #3,-(sp)
     trap        #14
     add.l       #2,sp
     move.l      d0,screen_address
```

```
* read
    move.l      #store_area,-(sp)    ; address of buffer
    move.l      #32000,-(sp)         ; buffer  size/number of bytes
* bytes to read
    move.w      handle,-(sp)
    move.w      #$3f,-(sp)
    trap        #1
    add.l       #12,sp
    tst.l       d0          ; see if error
    bmi         general_error

* close
    move        handle,(sp)
    move        #$3e,-(sp)
    trap        #1
    add         #4,sp
    tst.l       d0
    bmi         general_error

* wait for key press
    move        #2,-(sp)    ; device number (console)
    move        #2,-(sp)    ; BIOS routine number
    trap        #13         ; Call Bios
    addq.l      #4,sp

* vro_cpyfm()
* display DEGAS picture
    move        #109,contrl
    move.w      #4,contrl+2
    move.w      #1,contrl+6
    move.w      ws_handle,contrl+12
    move.l      #buffer,contrl+14         ; source address
    move.l      #screen_address,contrl+18    ; destination address
    move        #3,intin              ; replace mode

* source block coordinates
    move        #0,ptsin
    move        #0,ptsin+2
    move        #640,ptsin+4
    move        ht,ptsin+6
```

```
* destination coordinates
    move        #0,ptsin+8
    move        #0,ptsin+10
    move        #640,ptsin+12
    move        ht,ptsin+14
    jsr         vdi

* wait for key press
    move        #2,-(sp)      ; device number (console)
    move        #2,-(sp)      ; BIOS routine number
    trap        #13           ; Call Bios
    addq.l      #4,sp

exit:
*  close the virtual workstation
    move        #101,contrl
    clr.w       contrl+2
    clr.w       contrl+6
    move.w      ws_handle,contrl+12
    jsr         vdi

* appl_exit()
    move.l      #appl_exit,aespb
    bsr         aes          ; call AES

    move.w      #20,-(sp)  ; leave gracefully!
    move.w      #$4c,-(sp)
    trap        #1

* subroutines
general_error:
* a couple of examples
    cmpi.l      #-33,d0
    beq         error_message
    cmpi.l      #-34,d0
    beq         error_message
    bra         exit

error_message:
    move.l      #error,-(sp)   ; put address of string on stack
    move.w      #9,-(sp)       ; Gemdos function 'print a line',
```

```
*'Cconws'
        trap        #1
        addq.l      #6,sp         ; correct stack

* wait for key press
        move        #2,-(sp)      ; device number (console)
        move        #2,-(sp)      ; BIOS routine number
        trap        #13           ; Call Bios
        addq.l      #4,sp
        bra         exit

* AES subroutine
aes:    movem.l     d0-d7/a0-a6,-(sp)
        move.l      #aespb,d1
        move.l      #$c8,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

* VDI subroutine
vdi:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #vdipb,d1
        moveq.l     #$73,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

        ds.l    256
ustk:   ds.l    1

aespb:  dc.l    contrl,global,intin,intout,addrin,addrout
vdipb:  dc.l    contrl,intin,ptsin,intout,ptsout

* GEM arrays
contrl:             ds.w    12
intin:              ds.w    128
intout:             ds.w    128
global:             ds.w    16
addrin:             ds.w    128
addrout:            ds.w    128
```

```
ptsin:          ds.w   128
ptsout:         ds.w   128

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
graf_handle:    dc.w   77,0,5,0,0

error:          dc.b   'An error has occurred!',0
handle:         ds.w   1
file_name:      dc.b   'A:\MACART24.PI3',0
        even

pic_header:     ds.b 34
*MFDB's
buffer: dc.l    store_area      ; DEGAS pic
        dc.w    640
width:  dc.w    400
        dc.w    40
        dc.w    0
res:    dc.w    1
        dc.w    0
        dc.w    0
        dc.w    0

screen_address:  dc.l   1
        dc.w    640
wdth:   dc.w    400
        dc.w    40
        dc.w    0
res_:   dc.w    1
        dc.w    0
        dc.w    0
        dc.w    0

ht:             ds.w   1
p_handle:       ds.w   1
gr_handle:      ds.w   1
ws_handle:      ds.w   1
```

```
screen_res:     ds.w   1
     .bss
store_area:     ds.b 32000
```

One important feature of the program above is the use of the 'even' directive placed before the 'pic_header' array. This must be here as the address of 'pic_buffer' would be odd and the ST would crash when the program was run. This is because 'file_name' has an odd number of bytes in its buffer which produces an odd address for 'pic_buffer'. To circumvent this the 'even' directive is placed after 'file_name' to ensure that the next address is alligned correctly.

Most of the program uses many of the programming techniques discussed earlier in this book. However, an interesting feature is the method used to alter the MFDB values if the current screen resolution is medium. Initially the DEGAS picture is stored in a 32K buffer 'store_buffer:' with the MFDB structure set up for a high res DEGAS picture. However, by testing the current resolution we can then alter the values in the MFDB if necessary:

```
     cmp       #2,screen_res          ; is it high res
     beq       dont_alter_coords      ; no
* alter MFDB values
     move      #200,ht       ; med res screen height
     move      #200,width
     move      #200,wdth
     move      #2,res
     move      #2,res_

dont_alter_coords:
```

If you are using medium resolution the you should ensure that a '.PI2' file is used in the program, and 'file_name:' is altered appropriately.

By using the xor graphic mode it is possible to blit sections of the screen or GEM windows onto themselves and thereby clear the area very quickly. This can be very effective when a GEM window needs to be filled with a white fill after a redraw message is received. Taking the coordinates returned from 'evnt_multi' or from 'wind_set' instead of filling the window using 'vr_recfl', 'vro_cpyfm' can be used with excellent results.

Note that VDI clipping does not affect the VDI blitting operations, so great care has to be taken not to go beyond the screen boundary. Doing so may cause your program to crash as the VDI may write the bit image data to your actual program or program data area. This is because the screen image is held in screen RAM and your program may reside nearby.

excellent results.

Note that VDI clipping does not affect the VDI blitting operations, so great care has to be taken not to go beyond the screen boundary. Doing so may cause your program to crash as the VDI may write the bit image data to your actual program or program data area. This is because the screen image is held in screen RAM and your program may reside nearby.

Chapter 20
GDOS/ASSIGN.SYS

This chapter takes a detailed look at GDOS, and the ASSIGN.SYS file, and demonstrates how to get hard copy using GDOS and some VDI calls.

What is GDOS? GDOS is an acronym for Graphics Device Operating System and was left out of the ST's operating system ROM's. GDOS is an essential part of GEM, and specifically the VDI. Note that some VDI (Virtual Device Interface) calls to the operating system cannot be made without GDOS being installed. Doing so causes the ST to crash without warning. Some of these calls are:

v_opnwk	Open Workstation
v_clwk	Close Workstation
vst_load_fonts	Load GEM/GDOS fonts
vst_unload_fonts	Unload fonts
v_updwk	Update Workstation

Loading GDOS

GDOS is invariably loaded from an AUTO folder at boot–up. This means that a disk with an AUTO folder with 'GDOS.PRG' in it should be placed in drive A:/ when the ST is first switched on.

ASSIGN.SYS file

We have seen in chapter nineteen how to direct VDI calls to the screen, but what if we want to send the same output to the printer as well as the screen. To do this is where GDOS comes into the picture. With this booted we can open a workstation – a printer and direct output to it instead of the screen. But wait, there is more to it than that!

Another three essential items are first needed: an ASSIGN.SYS file, a printer driver and some fonts for any text output.

What is an ASSIGN.SYS file? An ASSIGN.SYS file goes hand in hand with GDOS, GDOS fonts, and printer output in general. It assigns or tells GEM what fonts we wish to use and what devices (eg printer, screen) we want the output sent to, and what printer driver we are going to use. As there are many different types of fonts, with various heights and styles the ASSIGN.SYS file allows us to tell GEM which specific fonts we want loaded with our application (DTP, or art program), and what particular device we are going to output to. In practise this invariably means screen and printer. However, note that there is low, med and hi res screens to chose from, and about 6 different types of printer, eg FX80 (standard Epson compatible 9-pin printer driver), LQ (standard Epson compatible 24-pin printer driver), etc.

Why use GDOS printer fonts? Because a printer is capable of a greater resolution than a monitor or tv. For instance the ATARI hi res monitor has a resolution of 90*90 dpi (dots per inch), whilst a standard 9-pin printer is capable of at least 120*144 dpi. Therefore to gain the best output possible it is essential to use printer fonts. Note that screen fonts and printer fonts should match in all aspects except of course, in size. A printer font will be much larger in size than its screen equivalent due to their differing resolutions, although when the printer font is used for output the screen representation and printed output should match, except the printed output should be finer. As GDOS fonts are bit-mapped, ie a character is made up of pixels that are either on or off, this effect is easy to see in a font editor.

The other essential that goes with GDOS, fonts, and general output is a printer driver. A printer driver is complex program that analyses the commands sent to it and acts upon them by sending appropriate commands to the printer for hard copy. A typical printer driver is the FX80.SYS program (for most 9-pin Epson compatible printers) which must be placed in the folder containing the fonts and printer driver. It is about 50K in size and note that a suitable printer driver is necessary for different types of printers, although the FX80 and LQ printer drivers are suitable for most popular 9 and 24 pin printers, including Star LC10, and Citizen 120d etc. Please note that this driver is not suitable for use with word processors like First Word.

Well that's enough theory! What about the practical side of GDOS? Note that GDOS fonts are also called GEM fonts.

It is essential that the ASSIGN.SYS file is on the same disk or partition if using a harddisk, as the GDOS.PRG, and in the root directory, ie not in a folder.

You also need the FONTS folder on the same disk, or on the disk pointed to by the path statement at the head of the ASSIGN.SYS file. The fonts folder contains the actual fonts used by the program (see later). The FONTS folder should contain GDOS screen fonts, printer fonts, and the printer driver, eg FX80.SYS.

If GDOS is successfully installed at boot-up then a message should come onto your screen (top left-hand corner) for a short period of time before you are returned to the ST's desktop. The message should read:

ATARI GDOS Ver. 1.1 resident.

If, however, GDOS was not installed other messages will appear briefly on the screen, for example telling you that the ASSIGN.SYS file contains an illegal workstation, ETC. You must correct this problem before proceeding.

Modifying the Assign.sys

To modify the ASSIGN.SYS file so that different fonts may be loaded other than the ones already specified in the file, you have to modify the file. This is achieved by loading the ASSIGN.SYS file into First Word and then turning off Word Processing (WP Mode – top of the EDIT menu). The file can then be modified to your purposes.

To change the printer driver you need only place the appropriate driver in the FONTS folder and change the name of the driver at 21, in the ASSIGN.SYS file.

As an example an ASSIGN.SYS file could contain the following text:

path=A:\FONTS\

```
;
01p screen.sys     ;Default screen
;
02p screen.sys     ;Low-resolution screen
;
03p screen.sys     ;Medium-resolution screen
;
04p screen.sys     ;High-resolution screen
;
COURIE10.FNT
COURIE14.FNT
COURIE18.FNT
COURIE28.FNT
COURIE36.FNT
;
21 FX80.SYS    ; EPSON FX80 and compatibles
EPSON__10.FNT
EPSON__14.FNT
EPSON__18.FNT
EPSON__28.FNT
EPSON__36.FNT
```

Do not leave a blank space in the ASSIGN.SYS file between the font names as this will cause the remainder of font names in the file to be ignored. Eg do not do this:

```
COMPUT16.FNT
COMPUT18.FNT

COMPUT28.FNT
```

You should not give any GDOS fonts a name starting with a number, eg 42NDST.FNT. This will cause GDOS not to be installed. In all events check your ASSIGN.SYS file carefully.

To use the fonts you want substitute the new font names for the present ones, or alternatively include your new choice of fonts with the present ones. Then save the new ASSIGN.SYS file and place on your working disk with the other files. Only have one ASSIGN.SYS file on your working disk.

Replace the fonts in the FONTS folder by the fonts specified in the ASSIGN.SYS file.

If you want to use your FONTS folder on another disk drive/partition you should alter the path statement in the ASSIGN.SYS file eg

Path = C:\FONTS\

You may call the folder what you will, so long as the name in the ASSIGN.SYS file corresponds with it.

Note that the ';' used in an ASSIGN.SYS file is there so that comments can be placed in the file. Any text on the same line as a ';' and placed after it will be ignored by GDOS, which is occasionally useful.

Fonts

Note that all fonts are loaded at start–up if there is enough RAM. If there is not enough RAM to accommodate all the fonts NO fonts will be loaded at all OR only some.

It is possible to have many different ASSIGN.SYS files, by naming them slightly differently from ASSIGN.SYS. So you could have 4 different ASSIGN.SYS files called ASSIGN.SY1, ASSIGN.SY2, ASSIGN.SY3 and the one you are using at start–up ASSIGN.SYS. By having different combinations of fonts in each ASSIGN.SYS file it is possible at start–up time by correctly renaming any one of the other three files to ASSIGN.SYS and renaming the other now unwanted ASSIGN.SYS to some other name to load different font styles.

Always reboot the computer if you change the ASSIGN.SYS file, or use a different printer driver.

Selecting a font size that is not available results in the next smaller size being used.

GDOS/GEM allows you to have a screen font half the printer font size. So, if you have a screen font 18 points high it is only necessary to have a printer font 9 points high to reproduce the screen font on your printer as the print program will double–up the printer font for you. This useful feature saves time and space.

On the disk you will find a simple ASSIGN.SYS file with some PD fonts in a FONTS folder with an FX80.SYS printer driver. Note that this will print-out on an Epson compatible 24-pin printer ok, although the scaling will be incorrect.

The ASSIGN.SYS file on the supplied disk:

```
; ASSSIGN.SYS FILE
path = \fonts\
;
01p screen.sys
;
02p screen.sys
;
03p screen.sys   ; MED RES SCREEN FONTS, New York
NEWYMD12.FNT
;
04p screen.sys   ; HI RES SCREEN FONTS
NEWYHI12.FNT
;
21 fx80.sys      ; EPSON 9-PIN PRINTER DRIVER AND
; PRINTER FONT
EPNEWY12.FNT
```

The next example source code demonstrates how to load a screen font and print to screen using this font rather than the normal system font.

```
* GDOS1.S
* Loads a GEM/GDOS screen font from disk and and displays the
* letter 'A' on screen, 12 points high. A rectangle is also
* displayed

*HEADER
    move.l      a7,a5
    move.l      #ustk,a7
    move.l      4(a5),a5
    move.l      12(a5),d0
    add.l       20(a5),d0
```

```
    add.l      28(a5),d0
    add.l      #$100,d0
    move.l     d0,-(sp)
    move.l     a5,-(sp)
    clr.w      -(sp)
    move       #$4a,-(sp)
    trap       #1
    add.l      #12,sp

* appl_intit()
    move.l     #appl_init,aespb
    jsr        aes ; call AES

* get current screen resolution
    move.w     #4,-(sp)
    trap       #14
    addq.l     #2,sp
    move.w     d0,res

* is gdos present
    moveq      #-2,d0
    trap       #2
    addq       #2,d0
    beq        no_gdos    ; or quit
    move       res,d0
    add        #2,d0
    move       d0,intin
no_gdos:
* if no GDOS should not continue

* graf_handle
    move.l     #graf_handle,aespb
    jsr        aes
    move       intout,gr_handle

* start by opening a virtual workstation
* v_openvwk
    move       #100,contrl
    move       #0,contrl+2
    move       #11,contrl+6
    move       gr_handle,contrl+12
```

```
    move       #1,intin+2
    move       #1,intin+4
    move       #1,intin+6
    move       #1,intin+8
    move       #1,intin+10
    move       #1,intin+12
    move       #1,intin+14
    move       #1,intin+16
    move       #1,intin+18
    move       #2,intin+20
    jsr        vdi
    move.w     contrl+12,ws_handle

* vst_load_fonts
* load fonts
    move       #119,contrl
    clr.w      contrl+2
    move.w     #1,contrl+6
    move.w     ws_handle,contrl+12
    move.w     #0,intin
    jsr        vdi

* vqt_name
* get name, font id, and style
    move       #130,contrl
    clr.w      contrl+2
    move.w     #1,contrl+6
    move.w     ws_handle,contrl+12
    move.w     #2,intin             ; second font 1=system font
    jsr        vdi
    move       intout,d0      ; get id only

* vst_font
* select actual font to use
    move       #21,contrl
    clr.w      contrl+2
    move.w     #1,contrl+6
    move.w     ws_handle,contrl+12
    move.w     d0,intin      ; id
    jsr        vdi
```

```
* vst_point
* set font height in points
    move        #107,contrl
    move.w      #0,contrl+2
    move.w      #1,contrl+6
    move.w      ws_handle,contrl+12
    move        #12,intin           ; height in points
    jsr         vdi

* v_gtext
* ouput graphic text
    move        #8,contrl
    move        #1,contrl+2
    move        #1,contrl+6   ; number of chars in string
    move        ws_handle,contrl+12
    move        #20,ptsin       ; x coord screen
    move        #50,ptsin+2   ; y coord screen
    move        #65,intin       ; actual character='A'
    jsr         vdi

* v_rbox
* ouput rounded rectangle
    move        #11,contrl
    move        #2,contrl+2
    move        #0,contrl+6
    move        ws_handle,contrl+12
    move        #8,contrl+10  ; function 8
    move        #100,ptsin  ; x coord screen
    move        #50,ptsin+2  ; y coord screen
    move        #100+60,ptsin+4 ; x coord right edge
    move        #50+40,ptsin+6 ; y coord bottom edge
    jsr  vdi

* wait for keypress (no echo)
    move        #8,-(sp)
    trap        #1
    addq.l      #2,sp

* vst_unload_fonts
* unload screen fonts
    move        #120,contrl
```

```
        clr.w       contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move.w      #0,intin
        jsr         vdi

* v_clsvwk
* close the virtual workstation
        move        #101,contrl
        clr.w       contrl+2
        clr.w       contrl+6
        move.w      ws_handle,contrl+12
        jsr         vdi

quit:
* appl_exit()
        move.l      #appl_exit,aespb
        bsr         aes    ; call AES

        move        #1,-(sp)
        move        #$4c,-(sp)
        trap        #1

aes:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #aespb,d1
        move.w      #$c8,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts
vdi:
        movem.l     d0-d7/a0-a6,-(sp)
        move.l      #vdipb,d1
        moveq.l     #$73,d0
        trap        #2
        movem.l     (sp)+,d0-d7/a0-a6
        rts

        ds.l 100
ustk:   ds.l 1
```

```
contrl:         ds.w 128
intin:          ds.w 128
intout:         ds.w 128
global:         ds.w 128
addrin:         ds.w 128
addrout:        ds.w 128
ptsin:          ds.w 128
ptsout:         ds.w 128

aespb:          dc.l contrl,global,intin,intout,addrin,addrout
vdipb:          dc.l contrl,intin,ptsin,intout,ptsout

p_handle:       ds.w 1
gr_handle:      ds.w 1
ws_handle:      ds.w 1
res:            ds.w 1
graf_handle:    dc.w 77,0,5,0,0
```

vst_load_fonts

The VDI call 'vst__load__fonts' trys to load the fonts from disk using the path specified by the ASSIGN.SYS file. It loads all the fonts specified in the ASSIGN.SYS too. It is not possible to load a particular font, all or none must be loaded. It returns in d0 the number of fonts loaded, if any. This is useful as a check can be made and if d0 contains 0 – the user should be informed that no fonts have been loaded, and that a disk containing the fonts should be placed in the disk drive. Similarly a check can be made for GDOS at the start of the program to see if it has been loaded and if it has not then the user should be informed and the program ended. Proceeding without GDOS will cause the program to crash.

The companion VDI call 'vst__unload__fonts' should be made when exiting to remove the loaded fonts from memory.

vqt_name

This VDI call is useful as it passes via the intout array information regarding the nature of the font. The system font is given the number 1, whilst all other loaded fonts are given font number starting at 2. As only one font has been loaded by the above program, as specified in the ASSIGN.SYS then it is safe to assume that the font we want to access is number 2, which is passed to the intin array. The intout array returns the font id (identification) number to be used in all subsequent calls in the first word of intout. The second word (intout+2) to the 64th (intout+64) word of the array contains the name of the font with each word containing the ASCII value of the name in the low byte and a null in the high byte, with the last 16 words containing the its thickness and style.

vst_font

This VDI function selects the font that will be used for all subsequent graphic text output. The font that we want to be set for use is the id number returned by the 'vqt_name' call in the first word of intout.

vst_point

This call sets the character height of the font currently in use to the height in points as specified in the value passed via the first word of the intin array. As not all heights are possible the VDI selects the next lowest available height.

v_gtext

This function outputs graphic text, and will output any ASCII character that has an available bit image in its character set/font. See chapter 19.

The next example program demonstrates how to redirect VDI output to the printer. It is virtually identical in all respects to the above program *except that a physical workstation (printer) is opened by 'v_opnwk' as* well as a virtual workstation (screen) by 'v_opnvwk'. Notice the similarity of the names. The physical workstation call is almost identical to the 'v_opnvk' except that the number 21 is passed to the intin array, which specifies the printer driver (FX80.SYS) named in the ASSIGN.SYS – this is then loaded by the call. The handle of the printer is passed via the contrl array (contrl+12) in the manner that the virtual workstation handle is received. This handle is then used in all the subsequent VDI graphic functions and the output is passed to the printer.

Note that the 'v_opnwk' intout array contains the maximum horizontal coordinate of the printer output in the first word of intout, and the second word contains the maximum vertical coordinate value, both in pixels. See disk for further information. If nothing is received in the first word of the intout array after calling 'v_opnwk' then we can be sure that no printer driver has been loaded and appropriate action can then be taken to inform the user.

```
* GDOS2.S
* Direct VDI output to a printer, using FX80.SYS 9-pin printer
* driver

*HEADER
    move.l      a7,a5
    move.l      #ustk,a7
    move.l      4(a5),a5
    move.l      12(a5),d0
    add.l       20(a5),d0
    add.l       28(a5),d0
    add.l       #$100,d0
    move.l      d0,-(sp)
    move.l      a5,-(sp)
    clr.w       -(sp)
    move        #$4a,-(sp)
    trap        #1
    add.l       #12,sp

* appl_intit()
    move.l      #appl_init,aespb
    jsr         aes     ; call AES

* get current screen resolution
    move.w      #4,-(sp)
    trap        #14
    addq.l      #2,sp
    move.w      d0,res

* is gdos present
    moveq       #-2,d0
    trap        #2
```

```
    addq    #2,d0
    beq     no_gdos
    move    res,d0
    add     #2,d0
    move    d0,intin
no_gdos:
* if no GDOS should not continue

* graf_handle
    move    #77,contrl
    move    #0,contrl+2
    move    #5,contrl+4
    move    #0,contrl+6
    move    #0,contrl+8
    jsr     aes
    move    intout,gr_handle

* v_opnvwk
* start by opening a virtual workstation
    move    #100,contrl
    move    #0,contrl+2
    move    #11,contrl+6
    move    gr_handle,contrl+12
    move    #1,intin+2
    move    #1,intin+4
    move    #1,intin+6
    move    #1,intin+8
    move    #1,intin+10
    move    #1,intin+12
    move    #1,intin+14
    move    #1,intin+16
    move    #1,intin+18
    move    #2,intin+20
    jsr     vdi
    move.w  contrl+12,ws_handle

* v_opnwk
* open printer
    move    #1,contrl
    move    #0,contrl+2
    move    #11,contrl+6
```

```
    move      ws_handle,contrl+12
    move      #21,intin           ; fx80 driver
    move      #1,intin+2
    move      #1,intin+4
    move      #1,intin+6
    move      #1,intout+8
    move      #1,intin+10
    move      #1,intin+12
    move      #1,intin+14
    move      #1,intin+16
    move      #1,intin+18
    move      #2,intin+20
    jsr       vdi
    move.w    contrl+12,p_handle   ; printer handle

*to see if there is there a driver, test d0, if zero no driver
    move      intout,d0       ; width
    move      intout+2,d1     ; height

* vst_load_fonts
* load fonts
    move      #119,contrl
    clr.w     contrl+2
    move.w    #1,contrl+6
    move.w    p_handle,contrl+12
    move.w    #0,intin
    jsr       vdi

* vqt_name
* get font id, name, and style
    move      #130,contrl
    clr.w     contrl+2
    move.w    #1,contrl+6
    move.w    p_handle,contrl+12
    move.w    #2,intin            ; second font 1=system font
    jsr       vdi
    move      intout,d0           ; get id only

* vst_font
* select actual font to use
    move      #21,contrl
```

```
    clr.w      contrl+2
    move.w     #1,contrl+6
    move.w     p_handle,contrl+12
    move.w     d0,intin
    jsr        vdi

* vst_point
* set font height in points
    move       #107,contrl
    move.w     #0,contrl+2
    move.w     #1,contrl+6
    move.w     p_handle,contrl+12
    move       #12,intin          ; height in points
    jsr        vdi

* v_gtext
* output graphics text
    move       #8,contrl
    move       #1,contrl+2
    move       #1,contrl+6   ; number of chars in string
    move       p_handle,contrl+12
    move       #20,ptsin       ; x coord printer
    move       #20,ptsin+2   ; y coord printer
    move       #65,intin       ; actual character='A'
    jsr        vdi

* v_rbox
* output rounded rectangle
    move       #11,contrl
    move       #2,contrl+2
    move       #0,contrl+6
    move       p_handle,contrl+12
    move       #8,contrl+10  ; function 8
    move       #100,ptsin      ; x coord printer
    move       #50,ptsin+2   ; y coord printer
    move       #100+60,ptsin+4 ; x coord right edge
    move       #50+40,ptsin+6 ; y coord bottom edge
    jsr        vdi

* update workstation- actually print
* v_updwk
```

```
        move        #4,contrl
        move.w      #0,contrl+2
        move.w      #0,contrl+6
        move.w      p_handle,contrl+12
        jsr         vdi

* vst_unload_fonts
* unload printer fonts
        move        #120,contrl
        clr.w       contrl+2
        move.w      #1,contrl+6
        move.w      p_handle,contrl+12
        move.w      #0,intin
        jsr         vdi

* v_clswk
* close the workstation (printer)
        move        #2,contrl
        clr.w       contrl+2
        clr.w       contrl+6
        move.w      p_handle,contrl+12
        jsr         vdi

* v_clsvwk
* close the virtual workstation
        move        #101,contrl
        clr.w       contrl+2
        clr.w       contrl+6
        move.w      ws_handle,contrl+12
        jsr         vdi

* appl_exit()
        move.l      #appl_exit,aespb
        bsr         aes        ; call AES

quit:
        move        #1,-(sp)
        move        #$4c,-(sp)
        trap        #1

aes:
```

```
         movem.l    d0-d7/a0-a6,-(sp)
         move.l     #aespb,d1
         move.w     #$c8,d0
         trap       #2
         movem.l    (sp)+,d0-d7/a0-a6
         rts
vdi:
         movem.l    d0-d7/a0-a6,-(sp)
         move.l     #vdipb,d1
         moveq.l    #$73,d0
         trap       #2
         movem.l    (sp)+,d0-d7/a0-a6
         rts

         ds.l 100
ustk:    ds.l 1

contrl:         ds.w 128
intin:          ds.w 128
intout:         ds.w 128
global:         ds.w 128
addrin:         ds.w 128
addrout:        ds.w 128
ptsin:          ds.w 128
ptsout:         ds.w 128

aespb:          dc.l contrl,global,intin,intout,addrin,addrout
vdipb:          dc.l contrl,intin,ptsin,intout,ptsout
p_handle:       ds.w 1
gr_handle:      ds.w 1
ws_handle:      ds.w 1
res:            ds.w 1
```

Chapter 21
Desk Accessories

This chapter looks at creating a simple desk accessory.

A desk accessory is a special type of GEM program that has the file extension '.ACC', and has to be booted from drive a:\ or from partition c:\ on a hard disk. Unlike other GEM and TOS programs (.PRG, .TOS, .TPP) it cannot be executed directly by double-clicking on it. A maximum of six desk accessories are normally available, however there are some programs available that can extend that number. A desk accessory is permanently installed (until removed by not booting with it) under the extreme left drop down menu, DESK and therefore is usable from any GEM program or the GEM desktop.

There are small but important differences between directly executable files and desk accessories. After starting the accessory by 'appl_init' which returns the application id number from intout, the desk accessory then installs itself in the DESK menu with the name passed to the addrin array, by using the 'menu_register' call. This returns the desk accessory menu identification number from the intout array, 'menu_id', which is used to identify it in any further operations. It then goes into a never-ending loop, see the 'wait' subroutine below, until it receives an 'ac_open' AES message as the user selects the DESK drop down menu. As the desk accessory has no control over the menu bar it cannot use the 'mn_selected' message in the way a usual application does so the AES sends an 'ac_open' message.

If the desk accessory is identified as the one we have implemented from the 'menu_id' value then a 'bsr' or 'jsr' to the desk accessory program proper is made. This obviously expects an 'rts' at some point, and your program should ensure that if the user signals an end to the use of the accessory it encounters one which will return it to the 'wait:' 'evnt_mesag' loop.

If an 'ac_close' message is received then it should be checked that the accessory window is open before doing the 'quit:' routine. This is because the 'evnt_mesag' routine can receive an 'ac_close' message if the user has called the accessory from within a main GEM application and is now closing down the application without first exiting the desk accessory. This is the reason for ensuring that the window handle 'w_handle' is passed a '-1' if no window is open. To go to the 'quit:' routine when the accessory window is not open is to invite all sorts of trouble.

Once the desk accessory is opened the program can be the same as any application and follows the same rules. Note that there is no 'pterm' call at the end as there is no standard GEM header file, although we must allocate ourselves a stack.

Note that there is no handling of redraw messages in the following code.

```
* ACC1.S
* This program should be assembled then the filename extension altered
* to .ACC
* It displays a GEM window on the screen.

        move.l      #ustk,a7
        move        #-1,w_handle
* appl_intit
        move.l      #appl_init,aespb
        jsr         aes         ; call AES
        move.w      intout,ap_rid

* graf_handle
        move.l      #graf_handle,aespb    ; get physical screen handle
        jsr         aes
        move        intout,gr_handle      ; store handle

        move        ap_rid,intin
        move.l      #menu_name,addrin
* menu_register
        move.l      #menu_register,aespb
        jsr         aes
        move        intout,menu_id
* if intout =-1 no room for another accessory
```

```
wait:
    move.l      #messagebuf_b,addrin
* evnt_mesag
    move.l      #evnt_mesag,aespb
    jsr         aes     ; call AES
    move.l      #messagebuf_b,a0
    move.w      (a0),d0 ; message type
    cmpi.w      #41,d0  ; close our window
    beq         our_window
    cmp.w       #40,d0
    bne         wait
    move.w      8(a0),d0
    cmp.w       menu_id,d0      ; open acc
    bne         wait
    bsr         do_it_
    bra         wait

our_window:
    move.w      6(a0),d0
    cmp         menu_id,d0
    beq         nr_quit
    bra         wait

nr_quit:
    cmp.w       #-1,w_handle
    beq         wait
    bra         quit

do_it_:
* start by opening a virtual workstation
    move        #100,contrl
    move        #0,contrl+2
    move        #11,contrl+6

* is GDOS present
    moveq       #-2,d0
    trap        #2
    addq        #2,d0
    beq         no_gdos         ; no GDOS
    move        res,d0
```

```
        add      #2,d0
        move     d0,intin

        bra      s_no_gdos
no_gdos:

        move     #1,intin            ; default if GDOS not
* loaded

s_no_gdos:
        move     #1,intin+2    ; line type
        move     #1,intin+4    ; colour for line
        move     #1,intin+6    ; type of marking
        move     #1,intin+8    ; colour of marking
        move     #1,intin+10   ; character set
        move     #1,intin+12   ; text colour
        move     #1,intin+14   ; fill type
        move     #1,intin+16   ; fill pattern index
        move     #1,intin+18   ; fill colour
        move     #2,intin+20   ; coordinate flag
        move.w   gr_handle,contrl+12  ; device handle
        jsr      vdi  ; v_opnvwk open virtual work station
        move.w   contrl+12,ws_handle  ; store virtual workstation handle

        jsr      mouse_off

* the type of the window
wtype            equ   $0ff

* the size lies in intout, so calculate the window size

* wind_get
        move.l   #wind_get,aespb
        move.w   #0,intin
        move     #5,intin+2
        jsr      aes

* wind_calc
        move     #1,intin
        move.w   #wtype,intin+2
        movem.w  intout+2,d0-d3       ; returned from wind get
```

```
	movem.w	d0-d3,intin+4		; the size
	move.l	#wind_calc,aespb
	jsr	aes

* now get its offsets
	move	intout+2,x
	move	intout+4,y
	move	intout+6,xwidth
	move	intout+8,ywidth

* and create the window
	move	#wtype,intin		; see above
	movem	intout+2,d0-d3
	movem	d0-d3,intin+2		; the size

* wind_create
	move.l	#wind_create,aespb
	jsr	aes
	move	intout,w_handle		; save the handle

* now set its title
	move.w	w_handle,intin
	move.w	#2,intin+2		; title string
	move.l	#windowname,intin+4	; the address
	clr.w	intin+8
	clr.w	intin+10

* wind_set
	move.l	#wind_set,aespb
	jsr	aes

	move.w	w_handle,intin
	move.w	#3,intin+2		; information string

	move.l	#info,intin+4

	clr.w	intin+8
	clr.w	intin+10

* wind_set
	move.l	#wind_set,aespb
```

```
        jsr         aes
* now actually show it by opening it
        move.w      w_handle,intin
        movem.w     x,d0-d3
        add.w       #5,d0              ; x start
        movem.w     d0-d3,intin+2      ; the size

* wind_open
        move.l      #wind_open,aespb
        jsr         aes

        jsr         wind_fill
        jsr         mouse_on
        jsr         arrow

        move.l      #messagebuf_b,addrin
e_multi:
        move.l      #evnt_multi,aespb

        move        #1+2+16,intin  ; keyboard, mouse, report
        move        #1,intin+2     ; number of clicks
        move        #1,intin+4     ; left mouse button
        move        #1,intin+6     ; left button down
        move        #1,intin+8     ; leave rect (not applicable)

        move        #0,intin+10
        move        #0,intin+12
        move        #0,intin+14
        move        #0,intin+16
        move        #0,intin+18
        move        #0,intin+20
        move        #0,intin+22
        move        #0,intin+24
        move        #0,intin+26
        move        #0,intin+28
        move        #0,intin+30
        jsr         aes

        move.w      intout,d0     ; 2=mouse 1= k/b, 16 = message
        cmpi.w      #$10,d0       ; message
        beq         mouse
```

```
        bra         e_multi
mouse:
        move.l      #messagebuf_b,a0
        move.w      (a0),d0
        cmpi.w      #$16,d0   ; L/Hand corner of window/close window
        beq         quit
        cmpi.w      #41,d0    ; acc close message
        beq         quit
        bra         e_multi

wind_fill:
* wind get
        move.w      w_handle,intin
        move        #4,intin+2
        move.l      #wind_get,aespb
        jsr         aes
        movem.w     intout+2,d0-d3

* bit blit
        move        #109,contrl
        move.w      #4,contrl+2
        move.w      #1,contrl+6
        move.w      ws_handle,contrl+12
        move.l      #Smfdb,contrl+14
        move.l      #Smfdb,contrl+18

        move        d0,ptsin
        move        d1,ptsin+2        ; was menu_ht

        move        d0,ptsin+8
        move        d1,ptsin+10

        add.w       d2,d0
        add.w       d3,d1
        sub.w       #1,d0
        sub.w       #1,d1

        move        d0,ptsin+4
        move        d1,ptsin+6

        move        d0,ptsin+12
```

```
        move      d1,ptsin+14
        move      #0,intin    ; ERASE SCREEN
        jsr       vdi
        rts

quit:

* wind_close
        move.w    w_handle,intin
        move.l    #wind_close,aespb
        jsr       aes

* wind_delete
        move.w    w_handle,intin
        move.l    #wind_delete,aespb
        jsr       aes

* close the virtual workstation
* v_clsvwk
        move      #101,contrl
        clr.w     contrl+2
        clr.w     contrl+6
        move.w    ws_handle,contrl+12
        jsr       vdi

* appl_exit()
        move.l    #appl_exit,aespb
        bsr       aes         ; call AES

        move      #-1,w_handle
        rts

        ds.l   100
        even
ustk:   ds.l   1

* subroutines

vdi:
        movem.l   d0-d7/a0-a6,-(sp)
        move.l    #vdipb,d1
```

```
        moveq.l   #$73,d0
        trap      #2
        movem.l   (sp)+,d0-d7/a0-a6
        rts

aes:
        movem.l   d0-d7/a0-a6,-(sp)
        move.l    #aespb,d1
        move.w    #$c8,d0
        trap      #2
        movem.l   (sp)+,d0-d7/a0-a6
        rts

mouse_off:
        movem.l   a0-a4/d0-d5,-(sp)
        dc.w      $a000
        move.l    4(a0),a1
        move.l    8(a0),a2
        dc.w      $a00a
        movem.l   (sp)+,a0-a4/d0-d5
        rts

mouse_on:
        movem.l   a0-a4/d0-d5,-(sp)
        dc.w      $a000
        move.l    4(a0),a1
        move.l    8(a0),a2
        clr.w     (a2)
        clr.w     2(a1)
        clr.w     6(a1)
        dc.w      $a009
        movem.l   (sp)+,a0-a4/d0-d5
        rts

arrow:
        move.l    #graf_mouse,aespb
        move      #0,intin
        jsr       aes
        rts
* end of subroutines
```

```
* keep these dc.w together
x:              ds.w 1
y:              ds.w 1
xwidth:         ds.w 1
ywidth:         ds.w 1

w_handle:       ds.w 1
ws_handle:      ds.w 1

windowname:     dc.b  'Example Window',189,0

vdipb:          dc.l   contrl,intin,ptsin,intout,ptsout
contrl:         ds.w   128
intin:          ds.w   128
intout:         ds.w   128
global:         ds.w   128
addrin:         ds.w   128
addrout:        ds.w   128
ptsin:          ds.w   128
ptsout:         ds.w   128

aespb:          dc.l   contrl,global,intin,intout,addrin,addrout

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
evnt_multi:     dc.w    25,16,7,1,0
wind_get:       dc.w    104,2,5,0,0
wind_calc:      dc.w    108,6,5,0,0
wind_create:    dc.w    100,5,1,0,0
wind_set:       dc.w    105,6,1,0,0
wind_open:      dc.w    101,5,5,0,0
graf_handle:    dc.w    77,0,5,0,0
graf_mouse:     dc.w    78,1,1,1,0
wind_close:     dc.w    102,1,1,0,0
wind_delete:    dc.w    103,1,1,0,0
menu_register:  dc.w    35,1,1,1,0
evnt_mesag:     dc.w    23,0,1,1,0
gr_handle:      ds.w    1

info:           dc.b  ' Information area:',0
smfdb:          dc.l   0              ; SCREEN
```

```
                ds.l   5
ap_id:          ds.w 1
messagebuf_b:   ds.b 16
                even
ap_rid:         ds.w   1
res:            ds.w   1
menu_name:      dc.b  '  Memo Taker',0
                even
menu_id:        dc.w   1
```

```
* keep these dc.w together
x:              ds.w 1
y:              ds.w 1
xwidth:         ds.w 1
ywidth:         ds.w 1

w_handle:       ds.w 1
ws_handle:      ds.w 1

windowname:     dc.b    'Example Window',189,0

vdipb:          dc.l    contrl,intin,ptsin,intout,ptsout
contrl:         ds.w    128
intin:          ds.w    128
intout:         ds.w    128
global:         ds.w    128
addrin:         ds.w    128
addrout:        ds.w    128
ptsin:          ds.w    128
ptsout:         ds.w    128

aespb:          dc.l    contrl,global,intin,intout,addrin,addrout

appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
evnt_multi:     dc.w    25,16,7,1,0
wind_get:       dc.w    104,2,5,0,0
wind_calc:      dc.w    108,6,5,0,0
wind_create:    dc.w    100,5,1,0,0
wind_set:       dc.w    105,6,1,0,0
wind_open:      dc.w    101,5,5,0,0
graf_handle:    dc.w    77,0,5,0,0
graf_mouse:     dc.w    78,1,1,1,0
wind_close:     dc.w    102,1,1,0,0
wind_delete:    dc.w    103,1,1,0,0
menu_register:  dc.w    35,1,1,1,0
evnt_mesag:     dc.w    23,0,1,1,0
gr_handle:      ds.w    1

info:           dc.b    ' information area',0
scrdb:          dc.l    0               ;SCREEN
```

Chapter 22 Miscellaneous

This chapter looks at common programming errors; useful programming utilities such as using the right mouse button with 'evnt_multi', finding the TOS version, booting from drive B:\, etc.

Programming errors

When calling a subroutine which uses the 'movem' instruction sometimes a branch out of the subroutine is made when there is an error code returned in the subroutine. For instance the example code below shows an open file routine. If an error is returned the error routine, 'general_error' is executed and the program branches back to the main part of the program, 'main', possibly an 'event_multi'. Unfortunately the stack remains uncorrected.

```
open:
     movem.l    d0-d7/a0-a6,-(sp)
     move.w     #0,-(sp)    ; set file attribute
     move.l     #file_name,-(sp)   ; address of filename
     move.w     #$3d,-(sp)     ; open function number
     trap       #1          ; hello GEMDOS
     add.l      #8,sp
     tst        d0              ; -ve number?
     bmi        general_error       ; yes, go to error routine
     move.w     d0,handle
     movem.l    (sp)+,d0-d7/a0-a6
     rts

general_error
* error code
     bra        main
```

Even when the 'movem' instruction is not used a programming error can occur:

```
open:
```

```
        move.w    #0,-(sp)        ; set file attribute
        move.l    #file_name,-(sp)   ; address of filename
        move.w    #$3d,-(sp)      ; open function number
        trap      #1              ; hello GEMDOS
        add.l     #8,sp
        tst       d0              ; -ve number?
        bmi       general_error     ; yes, go to error routine
        rts

general_error
* error code
    bra  main
```

Here the programmer is expecting to utilise the result of register d0, which returns the handle of the opened file. If an error occurs when opening the file the general error code will be executed. Once again the stack remains uncorrected, as an 'rts' was expected. This is the corrected code:

```
open:
        move.w    #0,-(sp)        ; set file attribute
        move.l    #file_name,-(sp)   ; address of filename
        move.w    #$3d,-(sp)      ; open function number
        trap      #1 ; hello GEMDOS
        add.l     #8,sp
        tst       d0              ; -ve number?
        bmi       general_error     ; yes, go to error routine
        rts

general_error
* error handling code
    rts
```

This is correct as the subroutine always reaches an 'rts' instruction which corrects the stack.

A very common error is to use the same name for a subroutine label and for a symbol constant by mistake.

```
evnt_multi:
        move.l    #messagebuf,addrin
```

```
        move.l    #evnt_multi,aespb
        move      #1+2+16,intin ; keyboard, mouse, report
        move      #1,intin+2    ; number of clicks
        move      #1,intin+4    ; left mouse button
        move      #1,intin+6    ; left button down
        move      #1,intin+8    ; leave rect (not applicable)

        move      #0,intin+10
        move      #0,intin+12
        move      #0,intin+14
        move      #0,intin+16
        move      #0,intin+18
        move      #0,intin+20
        move      #0,intin+22
        move      #0,intin+24
        move      #0,intin+26
        move      #0,intin+28
        move      #0,intin+30
        jsr       aes

        move.w    intout,d0     ; 2=mouse 1= k/b
        move.w    intout+2,mx   ; x mouse coord
        move.w    intout+4,my   ; y mouse coord
        cmpi.w    #$10,d0       ; mouse message
        beq       mouse
        cmpi.w    #2,d0    ; mouse button
        beq       evnt_multi


evnt_multi:     dc.w    25,16,7,1,0
```

The above program fragment illustrates the fault. When assembled no errors will be flagged, but when run and the 'evnt_multi' routine is entered, and a mouse button is pressed it is very probable that the program will branch to the dc.w label 'evnt_multi' which has the same name. The result will be an almost immediate crash. Of course the solution is easy:

```
e_multi:
    move.l    #messagebuf,addrin
    move.l    #evnt_multi,aespb
```

```
move      #1+2+16,intin  ; keyboard, mouse, report
move      #1,intin+2     ; number of clicks
move      #1,intin+4     ; left mouse button
move      #1,intin+6     ; left button down
move      #1,intin+8     ; leave rect (not applicable)

move      #0,intin+10
move      #0,intin+12
move      #0,intin+14
move      #0,intin+16
move      #0,intin+18
move      #0,intin+20
move      #0,intin+22
move      #0,intin+24
move      #0,intin+26
move      #0,intin+28
move      #0,intin+30
jsr       aes

move.w    intout,d0          ; 2=mouse 1= k/b
move.w    intout+2,mx    ; x mouse coord
move.w    intout+4,my    ; y mouse coord
cmpi.w    #$10,d0        ; mouse message
beq       mouse
cmpi.w    #2,d0         ; mouse button
beq       e_multi

evnt_multi:    dc.w    25,16,7,1,0
```

Take a look at the three separate program fragments below. In number three adding 99 to register a0 affects all of the a0 register which is what we would probably want. In the other two examples register d0 is only affected in the lower word, 1234 remains unaffected. This is potentially disastrous if an address is acted upon in d0 and then used later in the program. The address would refer to a place further back than when the address was acted upon by the 'add' instruction– a subtraction will have been the actual effect. Some very odd program behaviour can be expected then!

```
* 1
        move.l    #$1234ffff,d0
        addi.w    #99,d0        result: 12340062
* 2
        move.l    #$1234ffff,d0
        add.w     #99,d0        result: 12340062
* 3
        move.l    #$1234ffff,a0
        add.w     #99,a0        result: 12350062
```

Remember that when passing the address of a label that an address must always be accessed by a long word in length, so 'move.w #address,a0' will result in a program crash almost immediately. The correct way to pass an address is, of course, 'move.l #address, a0'. The 'address' label would have to refer to something like this:

```
address        dc.b    'Please place disk in drive',0

* or

address        dc.w    0,1,0,2,0
```

As we have discussed crashes so much in this chapter it is a good place to list the cause of crashes and the number of bombs (sometimes referred to as cherries) that each produces. The ST programmer soon becomes familiar with seeing bombs (meaning a crash has occurred) as a result of his or her programming.

Exceptions

The 68000 has a mechanism for handling severe programming errors called *exception handling* – the errors themseleves are known as exceptions.

When the programmer asks the 68000 to do something it cannot do an exception occurs and those bomb icons appear! They are in order of number of bombs:

Bus error (two bombs): this happens when the programmer tries to access memory that does not exist, or is protected from being accessed such as the ST's operating system. Certain addresses such as the system variables can only be accessed when in supervisor mode– see later. If a program that is not in supervisor mode tries to access these areas two bomb icons appear.

Address errors (three bombs): these occur when the programmer tries to reference a word or long on an odd byte boundary. this is why the 'even' directive is needed.

Illegal instructions (four bombs): these occur when the programmer tries to use an instruction that does not exist in the 68000's instruction set. At assembly time these errors would be caught by the assembler error trapping routine, but if your program accidentally overwrites a subroutine then when you came to access that subroutine you would probably see four bomb icons on screen as a result.

Zero divide (five bombs): this doesn't usually cause any bombs as TOS does not really consider this to be a serious error. This can occur when the programmer tries to divide something by zero.

CHK instruction (indexing errors) are caused by indexes of arrays becoming negative, or becoming bigger than the array. This gives six bombs

TRAPV instruction (overflow) is caused by a special instruction TRAPV. If two numbers are added and the result is too big to store then an overflow occurs. If a TRAPV instruction is placed after the ADD instruction whenever an overflow occurs seven bombs will be the result.

Privlege violations occur when the program tries to execute an instruction that is only allowed in supervisor mode. Eight bombs is the result.

Finding the TOS version:

Programming an application is made more difficult by the fact that there are a number of different TOS's, each having its own peculiarity. Often a programmer will find that a particular part of the program will function corectly with one TOS whilst it refuses to operate correctly with an earlier or later TOS. This can be a frustrating experience but one that has to be recognised by the programmer who wishes to sell his or her programs.

The following program whilst being useful in itself also illustrates the use of accessing a system variable. A system variable is an area of memory containing data that is guaranteed to remain consistent no matter what ATARI do with the machine. So a particular address that contains a pointer to another address which contains data useful to the programmer (and operating system) is guaranteed to remain inviolate. To access this area of memory it is necessary to switch to supervisor mode. Please see disk for a listing of the system variables.

```
* TOS_V.S
* This program returns the TOS version number from a system
* variable

* enter supervisor mode
    clr.l       -(sp)
    move.w      #$20,-(sp)
    trap        #1
    add.l       #6,sp
    move.l      d0,up_save    ; save user stack pointer

* get system base address, _sysbase

    move.l      $4f2,a0
    move.l      a0,sys_base   ; save for later use
    add.l       #$14,a0       ; add $14 to get '_os_magic' value
    move.l      (a0),a1
    move.l      (a1),d0
* should be 'magic' number #$87654321 in d0. This should be tested
* to confirm that we have a valid sys_base. Assume ok.

    clr.l       d0
    move.l      sys_base,a0
    add.l       #2,a0    ; add 2 to get actual TOS versions
    move.w      (a0),d0

* tos numbers 106= STE tos 1.6 with blitter
```

```
*       100= tos 1.0        ; (1985)
*       102= tos 1.2        ; + blitter (1987)
*       104= tos 1.4        ; (1988)

        cmpi.w      #$100,d0
        beq         tos1
        cmpi.w      #$102,d0
        beq         tos2
        cmpi.w      #$104,d0
        beq         tos3
        cmpi.w      #$106,d0
        beq         tos4
        bra         exit    ; TOS not recognised, could print erorr message
here

tos1
        move.l      #tos_v1,a0
        bra         cont

tos2
        move.l      #tos_v2,a0
        bra         cont

tos3
        move.l      #tos_v3,a0
        bra         cont

tos4
        move.l      #tos_v4,a0

cont:

        move.l      a0,-(sp)  ; put address of string on stack
        move.w      #9,-(sp)  ; Gemdos function 'print a line', 'cconws'
        trap        #1
        addq.l      #6,sp     ; correct stack

* wait for key press
        move        #2,-(sp)        ; device number (console)
        move        #2,-(sp)        ; BIOS routine number
```

```
    trap        #13        ; Call Bios
    addq.l      #4,sp

exit:
* restore user stack address, and exit supervisor mode
    move.l      up_save,-(sp)
    move.w      #$20,-(sp)
    trap        #1
    add.l       #6,sp

* pterm -exit cleanly
    move        #10,-(sp)
    move        #$4c,-(sp)
    trap        #1

up_save:        ds.l    1
sys_base:       ds.l    1

tos_v1:         dc.b    "TOS Version 1.00",0
tos_v2:         dc.b    "TOS Version 1.2",0
tos_v3:         dc.b    "TOS Version 1.4",0
tos_v4:         dc.b    "TOS Version STE 1.6",0
        even
```

Interrupt mouse handler

One if the drawbacks of using 'evnt_multi' is that we can set it up to recognize the press of the left mouse button but not the right button at the same time. Similarly we can set it up to recognize a right button event but not the left at the same time. This presents a dilemma as the programmer often wants to use both buttons at once. Many programs use this to good effect with a left mouse press signifying one particular choice whilst a right mouse button press signifies some other action the user wants the program to perform.

One solution is to use the VDI call 'vex_butv' call. Every time a mouse button is pressed we can pass a short routine to this call which it will run before GEM learns of the results of the button press. If the routine examines d0 it will find the result of the button press there, whether right, left, none, or both. If a right button is pressed we can pass a value to a symbol set aside for this purpose. At the same time we can place a one in d0 so that GEM thinks a left mouse button has been pressed. As this is all done under on an interrupt basis it is transparent to our main program. Now all we have to do is to set up 'evnt_multi' to look for a left button and when it drops through when we press the right mouse button we look at our holder to see whether a right mouse button has actually occurred.

The following routine sets this out in more detail.

```
* Install a new mouse handler that recognises
* the right mouse button using the vex_butv() VDI call.
* Mouse button status is same as vq_mouse() with values returned
*  in d0. 0=no button 1=left, 2=right, 3=both
* Every time the right mouse button is pressed 'button_state'
* is passed a value of 1, and 1 is passed to the handler to simulate
* a left mouse button. This ensures that when the right button is
* pressed evnt_multi sees it as a left button press and we can use
* evnt_multi as usual. We then examine button state to
* see whether the right mouse button was pressed when we have
* fallen through evnt_multi.


* install new mouse handler
    move        #125,contrl
    move.w      #0,contrl+2
    move.w      #0,contrl+6
    move.w      gr_handle,contrl+12     ; physical screen device handle
    move.l      #new_mouse,contrl+14 ; address of new mouse handler
    jsr         vdi

    move.l      contrl+18,old_mouse_addr

* de-install new mouse handler
    move        #125,contrl
    move.w      #0,contrl+2
    move.w      #0,contrl+6
    move.w      gr_handle,contrl+12
    move.l      old_mouse_addr,contrl+14
    jsr         vdi

new_mouse:
```

```
    cmpi.w  #2,d0       ; right button
    bne.s   nowt
    move    #1,button_state
    moveq   #1,d0       ; pretend it was left button
nowt:
    move.l  old_mouse_addr,-(sp)
    rts

button_state:    ds.w  1
old_mouse_addr ds.l 1
```

The 'new__mouse' routine should be installed at the start of our application. The old address of the routine can be found from 'contrl+18' and is placed in 'old__mouse__addr' for safe keeping. When our application exits back to the desktop we should de-install our new mouse handler and pass it the old mouse address via 'old__mouse__addr'.

Now when 'evnt__multi' is used – it should be set up to recognize a left mouse button event – and if the right mouse button is pressed it will think that left mouse button event has occurred. We should then examine 'button__state' to see if a right mouse button was pressed and then act accordingly. The value of button__state should always be set to zero after it has been examined after every 'evnt__multi' call, 'clr.w button__state'.

Booting from drive B:\

It is often very useful to be able to boot from drive B:\, especially if you own an early ST with a single-sided drive, and have an external double-sided drive. A lot of software is provided on double-sided disks and often needs to be booted from drive a:\ – the internal drive. The following program allows the user to boot from drive B:\ by bypassing the need to boot from the internal drive.

After the program has been assembled and run the ST should be reset and until the ST is switched off drive B:\ will be the boot drive.

```
* B_BOOT.S
* This program allows the user to boot from drive B:
```

```
* print message
    move.l      #message,-(sp)  ; put address of string on stack
    move.w      #9,-(sp)  ; Gemdos function 'print a line', 'cconws'
    trap        #1
    addq.l      #6,sp     ; correct stack

* enter supervisor mode
    clr.l       -(sp)
    move.w      #$20,-(sp)
    trap        #1
    add.l       #6,sp
    move.l      d0,up_save    ; save user stack pointer

* make boot device B, using system variable $446

    move.w      #1,$446   ; drive b now boot drive, 0=drive a:\

* wait for key press
    move        #2,-(sp)      ; device number (console)
    move        #2,-(sp)      ; BIOS routine number
    trap        #13           ; Call Bios
    addq.l      #4,sp

exit:
* restore user stack address, and exit supervisor mode
    move.l      up_save,-(sp)
    move.w      #$20,-(sp)
    trap        #1
    add.l       #6,sp

* pterm -exit cleanly
    move        #10,-(sp)     ; exit code
    move        #$4c,-(sp)
    trap        #1

up_save:        ds.l    1

message:    dc.b    'Set drive B to be boot disk. Reset ST afterwards',13,10
                dc.b    'Press a key to continue',0
                even
```

Hex to ASCII

The programmer often has to report to the user of his/her software certain conditions that have been requested by the user. For example it is often useful for the user to know how much free RAM is available. This information is easy to get with a simple call to the o/s. However, a difficulty arises here as the information is given to us in hexadecimal whilst we need it in ASCII format. For instance if we had the amount of free RAM in register d0 and then decided we wanted to print this to the screen we would find that the hex values would be interpreted as ASCII values and therefore of little use. Experiment your self and you will soon see what I mean. What we need is a utility that would alter the hex values to ASCII for us so that when we printed them to the screen we would see the correct amount.

To demonstrate this principle and to provide a useful program please examine the following source code.

```
* FREE_RAM.S
* Get amount of free RAM, and convert hex amount to ASCII
* Display amount to screen

* header
        move.l      a7,a5
        move.l      #ustk,a7
        move.l      4(a5),a5
        move.l      12(a5),d0
        add.l       20(a5),d0
        add.l       28(a5),d0
        add.l       #$100,d0
        move.l      d0,-(sp)
        move.l      a5,-(sp)
        clr.w       -(sp)
        move        #$4a,-(sp)
        trap        #1
        add.l       #12,sp

* get free RAM
        move.l      #-1,-(sp)
        move        #$48,-(sp)
        trap        #1
```

```
    addq.l     #6,sp      ; free RAM returned in d0
    clr.l      d1
    move.l     d0,d1
    clr.l      d0
    clr.l      d3

* hex to ascii
    move.l     #ram_amount+10,a6 ; address of where amount will be
    jsr        convert        ; after converting
    move       #10,d4
    move.l     #ram_amount,a2
    jsr        check_spaces    ; alter preceeding zeros to spaces
    move.l     #ram_amount,-(sp) ; put address of string on stack
    move.w     #9,-(sp)       ; Gemdos function 'print a line'
    trap       #1
    addq.l     #6,sp

* wait for key press
    move       #2,-(sp)       ; device number (console)
    move       #2,-(sp)       ; BIOS routine number
    trap       #13            ; Call Bios
    addq.l     #4,sp

exit:  move.w  #20,-(sp)  ; leave gracefully!
    move.w     #$4c,-(sp)
    trap       #1

convert:
    move.l     #10,d2
another_num:
    move.w     d1,d3
    clr.w      d1
    swap       d1
    divu       d2,d1
    bvc        skip
    rts

skip:  move.w  d1,d4
    move.w     d3,d1
    divu       d2,d1
    swap       d1
```

```
        addi.b      #$30,d1
        cmpi.w      #'0',d1
        blt         here
        cmpi.w      #'9',d1
        bgt         here
        move.b      d1,-(a6)
here:   move.w      d4,d1
        swap        d1
        tst.l       d1
        bne         another_num
        rts

check_spaces:
        clr.l       d3
checkagain:
        move.b      (a2)+,d3
        subi.w      #1,d4
        cmpi.w      #0,d4
        beq         checkfinished
        cmpi.b      #48,d3
        blt         put_space
        bra         checkagain

put_space:
        suba        #1,a2
        move.b      #' ',(a2)+
        bra         checkagain

checkfinished:
        rts

        ds.l   50
ustk:   ds.l   1

ram_amount:   ds.l   4
```

Once we have the amount of free RAM in register d0 we need to convert this hex number to its equivalent ASCII representation. To do this we need to convert the free RAM to ASCII by placing it in a string which then can be printed by 'cconws', or something similar. The rou–

tine 'convert:' expects the hex amount to be in register d1, and an empty string address in register a6 so that it can place the converted hex values there.

Once the ASCII values are placed in the string 'ram_amount' then we need to place spaces before any ASCII numbers that are preceded by nulls. The routine 'check_spaces:' does this. This is necessary as GEM sees a null as an end of string marker and if we try to print a string that has a null at the start of it we will find that nothing will be printed.

Note that the 'convert:' routine needs to place the converted ASCII values in the ram_amount string from the end of the array. If this was not done the result would be an ASCII string back-to-front! This is the reason for the *+10* in the statement 'move.l #ram_amount+10,a6'.

The 'convert:' and 'check_spaces:' routines are general purpose routines that can be easily adapted for use in your own software.

ASCII to hex

Another problem that the programmer is faced with is converting ASCII input to hex. For instance a user may input some figures in a dialog box, say for setting margins in a word processor. the result is an ASCII string that needs converting to hexadecimal before anything can be done with it.

In the following code we simulate input via a dialog box by placing the number 3, ASCII code 51 in 'amount'. The rest of the program concerns itself with converting this ASCII code to a hex number.

Once again the string passed to the conversion routine has to be the end of the array, so that ASCII code can be picked off starting with the lowest part of the number.

The 'mult:' routine is a special routine that can be used separately to give greater accuracy when multiplying two large numbers together than the mulu instruction alone.

```
* ASC_HEX.S
* Get ASCII number from user, probably dialog box and convert to
```

```
* hex for use within program

        move.b   #51,amount   ; place number '3' in string

        move.l   #4+1,d5       ; number of bytes +1, in 'amount'
        move.l   #amount+4,a2  ; address of end of string in a2
        jsr      convert       ; hex value is returned in d6

exit:   move.w   #20,-(sp)  ; leave gracefully!
        move.w   #$4c,-(sp)
        trap     #1

convert:
        clr.l    d0
        clr.l    d6
        move.l   #1,d2
again:  move.b   -(a2),d1
        subi     #1,d5
        beq      ret
        cmp.b    #'0',d1
        blt      again
        cmp.b    #'9',d1
        bgt      again
        and.l    #$000f,d1
        jsr      mult
        add.l    d1,d6         ; D6 contains result
        move.l   #10,d1
        jsr      mult
        move.l   d1,d2
        bne      again
ret:
        rts
mult:
* mult d1 and d2 return result in d1
        movem.l  d2-d4,-(sp)
        move.w   d1,d3
        move.w   d2,d4
        swap     d1
        swap     d2
        mulu     d3,d2
        mulu     d4,d1
```

```
    mulu        d4,d3
    add.w       d2,d1
    swap        d1
    clr.w       d1
    add.l       d3,d1
    movem.l     (sp)+,d2-d4
    rts

amount:         ds.l    1
```

Chapter 23
Using the Text Editor

This chapter deals with the use of the editor – EDITOR.PRG, and assembling and debugging programs from it, but does not go into detail about the assembling/debugging process. See the next chapter for a detailed look at the assembler and debugger.

The text editor on the disk supplied with this book is specially written to be used in conjunction with the book.

A text editor differs from a word processor in that a text editor permits none of the text attributes such as bold, underline and italic, etc common to such word processors such as 1ST Word. Word wrap or any type of justification does not exist. However, none of these particular functions are needed when writing assembly language programs.

All other functions are very similar to a word–processor. zzSoft's text editor is very similar to HiSoft's DEVPAC text editor. If you are familiar with word processing software, the supplied text editor will hold few surprises for you.

The main use of the editor is to enter and edit assembly language programs (source code), store the text on disk, and load the text back again when necessary, and assemble the text into either executable programs and/or object files that can be linked into executable programs.

To list the possibilities:

Editable source code to: .PRG files (executable program)
.O files that can be linked to give .PRG files.

The editor acts as a shell from which source code can be assembled and debugged without leaving the editor. The current assembled executable file can also be run without leaving the editor. The only time you might need to leave the editor is when a program crashes and the ST locks–up or is unusable and you have to reboot.

Dialog boxes

The editor makes extensive use of dialog boxes which are boxes that can have information with buttons, radio buttons, and editable text. For instance pressing the *HELP* key will bring a dialog box to the screen presenting you with box of helpful text. There is also a button with the word 'Ok' written inside. This 'Ok' button is used to tell the dialog box that you have finished with it by either clicking inside it with the mouse pointer or pressing the Return key– which simulates the mouse click. Sometimes there are choices such as 'Ok' and 'Cancel'. By clicking in 'Cancel' this will cancel anything you may have done in the dialog box, such as entering text, and return it to the state it was in before it was invoked. This will also signal to the program that you have finished with it. If a button has a thicker border than any others then this button is said to be the 'default' and this button can also be selected by pressing the Return key..

If a button is in inverse video, ie black with white text, this means that it is, or has been, selected.

Radio Buttons

Radio buttons are groups of buttons. Selecting one, which makes it appear black with the text white (inverse video), immediately deselects any in that group, if any had been selected. These buttons are usually grouped in such a way as it would be expected that such action would take place. For instance in the Assembly dialog box you can select either a '.PRG' or '.O' file but not both.

Editable text in a dialog box is shown with a dotted line and a cursor that is a thin vertical line. Characters can be entered and corrected by using the backspace, delete, and Esc key which clears the whole line. The cursor keys can be used to position the cursor. Some dialog boxes only allow a limited range of characters such as the *Goto Line...* dialog box which only allows numbers to be entered.

When a button or text (usually in a menu option) is greyed out then that part of the dialog box is not selectable.

Entering text, using the cursor, and mouse pointer.

As soon as the editor is loaded you will be presented with an empty screen, a still cursor in the top left-hand corner, and a status line, topped by the menu bar. See diagram 23:1 below which shows the editor with the *File* drop down menu selected.

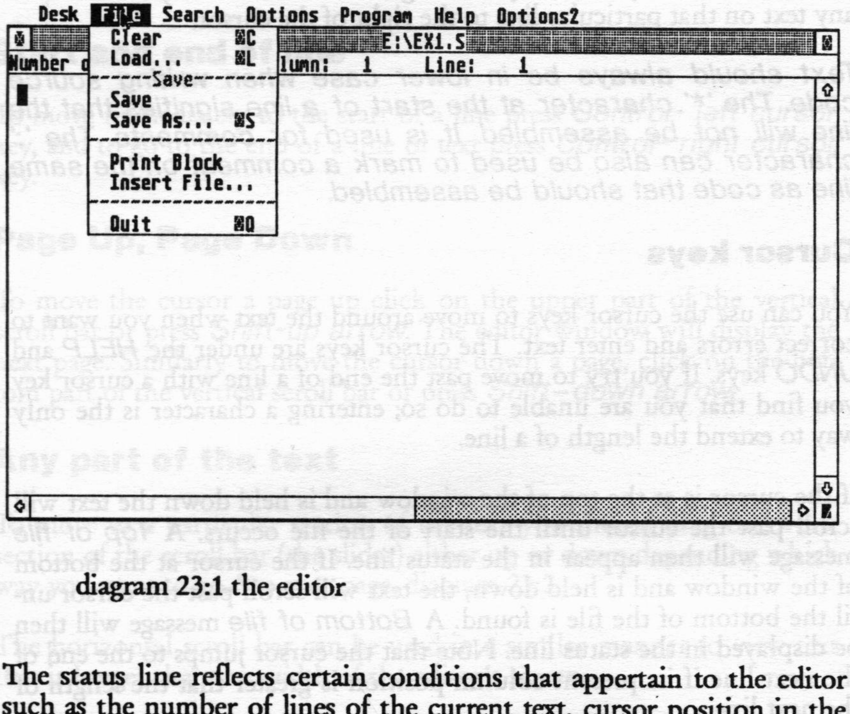

diagram 23:1 the editor

The status line reflects certain conditions that appertain to the editor such as the number of lines of the current text, cursor position in the form of line and column. The right hand side of the status line is used to display useful messages such as 'bottom of file' etc. If text is present in the editor the path and file name is shown above the status line – in the shaded area.

To enter text you just type at the ST's keyboard and whatever you type is shown on screen. This is known as echoplexing or echoing for short. As you press a key the cursor will advance along the line. As you reach the end of a line you should press the Return key which will take you to the next line. If you enter long lines of text the window will scroll sideways. You can correct mistakes by pressing the backspace key which deletes the character to the left of the cursor, or delete key which removes the character the cursor is over. Holding down the backspace key will result in the cursor moving left until it meets the left hand part of the screen and cannot go any further. It then will go to the next line of text above (if there is one) and start to remove any characters it meets there. You can use the backspace key to join text from a line to the one above it in this way. Similarly holding down the delete key will remove any text on that particular line to the right of the cursor.

Text should always be in lower case when writing source code. The '' character at the start of a line signifies that the line will not be assembled. It is used for comments. The ';' character can also be used to mark a comment on the same line as code that should be assembled.*

Cursor keys

You can use the cursor keys to move around the text when you want to correct errors and enter text. The cursor keys are under the *HELP* and *UNDO* keys. If you try to move past the end of a line with a cursor key you find that you are unable to do so; entering a character is the only way to extend the length of a line.

If the cursor is at the top of the window and is held down the text will scroll past the cursor until the start of the file occurs. A *Top of file* message will then appear in the status line. If the cursor at the bottom of the window and is held down, the text will scroll past the cursor until the bottom of the file is found. A *Bottom of file* message will then be displayed in the status line. Note that the cursor jumps to the end of the next line if its present column position is greater that the length of the next line.

You can also move the cursor by (repeated) clicking on the arrow boxes at the end of the horizontal and vertical scroll bars.

Mouse pointer positioning

You can also position the cursor at any character by placing the mouse pointer over a character and clicking the left mouse button. The cursor will immediately jump to that position. Trying to position the cursor with the mouse pointer anywhere other than over a character will result in it being placed at the end of the nearest line of text. You can also select a block of text by holding down the left mouse button and drawing an outlined box over the text you wish to select. This text will be then shown in reverse video (white text on a black background) and may then be deleted or moved to another position (copied) by selecting *Delete* or *Paste* from the Options menu or *Shift-F4* or *F4* respectively. See later.

Start and end of line

To move immediately to the start of a line press *Control- left cursor* key, and to go to the end of a line of text press *Control- right cursor* key.

Page Up, Page Down

To move the cursor a page up click on the upper part of the vertical scroll bar or press *Shift-up arrow.* The editor window will display the next page. Similarly to move the cursor down a page, click on the bottom part of the vertical scroll bar or press *Shift–down arrow.*

Any part of the text

To move to a particular section of the text you should drag the white section of the scroll bar (the slider) either up or down depending which way you want to go. See over page, diagram 23:2

The horizontal scroll bar can be used in a similar manner to view text that is longer than the width of the editor's screen.

The Tab key

The Tab key will move the cursor ten spaces and is useful for tabulating your source code neatly.

A maximum number of 150 characters (including spaces) are allowed per line.

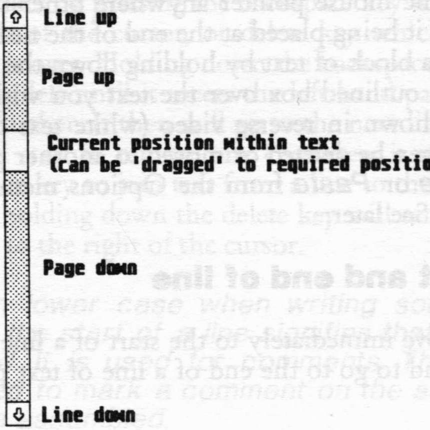

diagram 23:2

Goto a line

To move the cursor to any particular line of text select *Goto line...* from the Options drop down menu and after the dialog box has appeared enter a number. To go to the required line press the *Return* key or click in the *Ok* button. To abort click in the *Cancel* button.

Goto top of file

To move to the start or top of text file press *ALT-T* or click on *Goto Top* from *Options* menu. The cursor will be placed at line 1.

Goto end of file

To get to the last line or bottom of your text file press *ALT-B* or click on *Goto Bottom* from *Options* menu.

Quitting the editor

Press *ALT-Q* to leave the editor or select *Quit* from the *File* menu.

Deleting text

Delete a line

The line the cursor is positioned on can be removed by pressing *Control-Y.*

Undelete a line

The last line that has been deleted with *Control-Y* can be reinserted into the text by pressing *Contol-U.*

Delete all of the text

Selecting *Clear* from the *File* drop down menu will remove all text from the screen and from the text editor. This means that unless you have saved your text to disk it cannot be recovered.

Using the mouse to select text for deletion or pasting

Text in the GEM window may be selected by pressing the left mouse button down at the cursor position and drawing an outlined box around the text you want selected. The text thus selected will be then shown in reverse video, and various operations can then be carried out. Immediately the text is marked as a block in reverse video it is placed in an internal buffer and thus can be deleted, or pasted to any position within the file. The *delete* and *paste* options are selected from the *Options* drop down menu. Alternatively you may use the Function key options for deleting (*shift F4*) and pasting (*F4*) text

If a block of text is highlighted and this includes all the text up to and including the last line and then deleted but the text does not occupy more than a full screen so that the right–hand scroll bar empty it can seem that all the text has been deleted. However, this is not the case as using the cursor–up key to display the rest of the text will demonstrate. The reason for this seemingly silly behaviour is that when a block of text is deleted the cursor is placed at the first line of the deleted block of text.

Disk operations:

GEM file selector

The GEM file selector is a standard dialog box which is normally used to facilitate the loading and saving of files. The diagram over the page shows its main features. Please note the TOS 1.4 and STE dialog box has been improved to include disk drive buttons.

Saving text (ALT–S)

To save the text file currently held in the text editor press ALT–S or select *Save as...* from the *File* drop down menu to bring the GEM file selector box on screen. A suitable file name should be entered on the command line or a file name selected with the mouse pointer which will place it on the command line to be altered if necessary. Press the *Return* key, or click in the *OK* box to complete this operation. Click in the *Cancel* button to abort process. The contents of that file (if it already exists) will be over–written with the present contents of the text editor. Double clicking on a displayed file name will result in the same action.

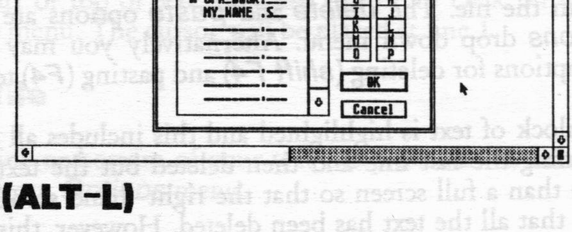

diagram 23:3
File selector

Loading text (ALT–L)

Loading text follows a similar procedure to that above. Note that any text in the editor will be completely over–written and cannot be recovered unless previously saved to disk.

The file name and path of the loaded or saved text is displayed above the status line, and is updated if and when the file is saved with a different name or a different file is loaded.

Inserting text file (ALT-I)

To insert a text file at any point in your text you should position the cursor where you want the text inserted. Press *ALT-I* and you will be presented with the GEM file selector. Assuming that you do not select cancel the text file you select will be read from disk and inserted. If there is not enough memory to permit this a warning will be given.

Searching and Replacing Text

Find

If you wish to find a particular word or some text in your text you should press *ALT-F* or go to the *Search* drop down menu and select *Find...*

You should enter the string you want to find when the dialog box appears in the 'Find' field. If you press Return or click in the *Next* button the search for your string will start forwards. If the string can be found the program will place the cursor on the text. If no text can be found the cursor will remain stationary. If you click in the *Previous* button the search will begin backwards. To search for more occurrences of the string you should press *ALT-N* to go forwards or *ALT-P* to go backwards.

Replace

If you enter text in the 'Replace' field and the program finds the text entered in the 'find' field, this text will be replaced with the text entered in the 'replace' field. To continue with replacing press *ALT-N* for the next string or *ALT-P* for the previous (if any).

Replace all

To replace all the occurrences of the string you should place the cursor at the top of the file and select *Replace All* in the *Search* drop down menu. Immediately the program will start to replace the text with your string. As the program does this it will indicate its progress by moving the vertical scroll bar position indicator – if sufficient text is held in the editor. When all the text has been replaced the number of text replacements is reported.

Block commands

A block of text is a marked section of text that can be deleted, copied, saved to disk, or printed. If a block of text cannot be found by the program the error message 'What Blocks!' is given.

Marking a Block (F1, F2)

To mark a block of text use the *F1* key to mark the start of the block at the cursor position. To mark the end of the block you should move the cursor to the required position and press *F2* Both start and end of a block are marked with arrows. Note that the end of a block should be marked on the line after the actual line.

Saving a Block (F3)

To save a block of text to disk you should press *F3* and enter the name of the file in the GEM file selector.

Deleting a Block (Shift-F4)

To delete a block of text you should press *Shift-F4*. The block can be restored by pressing *F4*.

Pasting a Block (F4)

To paste or copy a block of text you should place the cursor at the position in the text were you want the text to be copied to and press *F4*.

Printing a Block

To print the current marked block of text you should select *Print* *block...* from the *File* drop down menu.

Miscellaneous Commands

Help (ALT-H, HELP)

Pressing the *HELP* key or pressing *ALT-H* or selecting from the *HELP* drop down menu will bring a dialog box on screen containing some useful reminders.

Assembler Help

Pressing *ALT-E*, or selecting from the *HELP* drop down menu will bring this dialog box on screen.

Editor Help

As above.

Touch

This option allows files to be updated to the current time and date.

Delete a file

This option allows the user to delete files from the disk. Warnings are given.

Format a disk

This useful option allows you to format disk in drive A, either single-sided or double-sided. Warnings are given to prevent accidental formatting of disks. A standard format is used.

Text editor error messages

I can't find the resource file- Exit

When double-clicking on the text editor program 'EDITOR.PRG' to run it the first action it takes is to try and load its resource file 'EDITOR.RSC'. If the resource file cannot be found then the program cannot continue as it is essential that the editor loads this file. This usually happens when the executable file is copied from one disk to another and the resource file is not copied along with it. The solution is to ensure that the resource file EDITOR.RSC file is always with the EDITOR.PRG.

Many of the error messages reported to the user are system messages passed via GEMDOS to the program and these are the type of errors that might occur if a disk is corrupted, or if you try to save a file to full disk or when there is no disk in the drive.

Other error messages that the editor may report are those that result when a file cannot be found when assembling or linking:

File (ASSEMBLR.TTP) not found

This usually means that the path has not been correctly set in the *Set Paths...* drop down menu selection dialog box, or that the file is not on the disk. Other files that may suffer from the a similar message might be the linker 'LINK.TTP', or the debugger.

Insufficient memory to run ASSEMBLR.TTP

This message means that there is not enough free RAM memory to run the assembler, and probably means that you have too many desk accessories loaded. Solution: exit editor and reboot with fewer accessories. This message may also be received about the linker, executable files, and the debugger.

Can't insert a line, or block as maximum number of lines (1000) reached. Try deleting some lines first.

The text editor can only support 1000 lines of text of a maximum length of 150 characters. So this error message means that the editor is full and cannot support any further text. If some lines of text are deleted or a block saved to disk and then deleted then the block or line can the be inserted. The maximum length of each line of text allowed is 150 characters including spaces.

What blocks!

This message results if a block has not been marked out correctly or not at all. When marking out a single line of text the second marker should be on the next line.

File to big to insert! Would cause text buffer overflow.

When trying to insert a file to disk it may not be apparent that if the file was inserted then the file buffer would overflow. That is, more than 1000 lines of text would be the result of inserting the file into the displayed text file which is the maximum allowed. To insert the file would result in the ST crashing very quickly so this is not allowed.

Converting the source files for use with HiSoft's DEVPAC

To convert the example source files for use with this popular assembler development package the '.bss', and '.data' directives should be altered to SECTION BSS and SECTION DATA. Note that DEVPAC does not support the '.globl' directive, although labels can be dumped easily.

For earlier versions of DEVPAC the '.bss' directive should be altered like this:

from

```
    .bss
buffer  ds.b    32000
```

to

```
buffer  dsbss.b 32000
```

Keyboard Options, a complete listing:

ALT–C	Clear	(remove all the text)
ALT–L	Load	(a text file)
ALT–S	Save as	(save a text file via file selector)
ALT–Q	Quit	(leave EDITOR.PRG and go to GEM desk-

top)

ALT–F	Find	(a word or phrase)
ALT–N	Find next	(find next occurrence of word or phrase)
ALT–P	Find previous	(occurrence of word or phrase)
ALT–R	Replace	(replace word with word or phrase or nothing)
ALT–G	Goto line	(place cursor on line specified)
ALT–T	Goto top	(go to top of file–line 1)
ALT–B	Goto bottom	(go to bottom of file)
ALT–A	Assembler	(invoke assembler dialog box)
ALT–X	Run	(run last executable file assembled)
ALT–D	Debug	(invoke debugger with last executable file)
ALT–J	Run Debug	(invoke debugger)
ALT–O	Run other	(run any other executable program, selected from file selector)
ALT–H	Editor Help	(show editor help dialog)
ALT–E	Assembler Help	(show assembler help dialog)

Chapter 24
Using the Assem'r & Debug'r

Although explanation of the use of the assembler and debugger is provided throughout the book this chapter provides further detailed descriptions of the assembler and debugger's operation and methods of use.

The assembler

The assembler and linker are separate programs from the text editor and can be found on the disk as ASSEMBLR.TTP and LINK.TTP. The assembler and linker must be both on the same disk and in the same folder, if any, as this is expected by the text editor which searches for these files using the same path for both files.

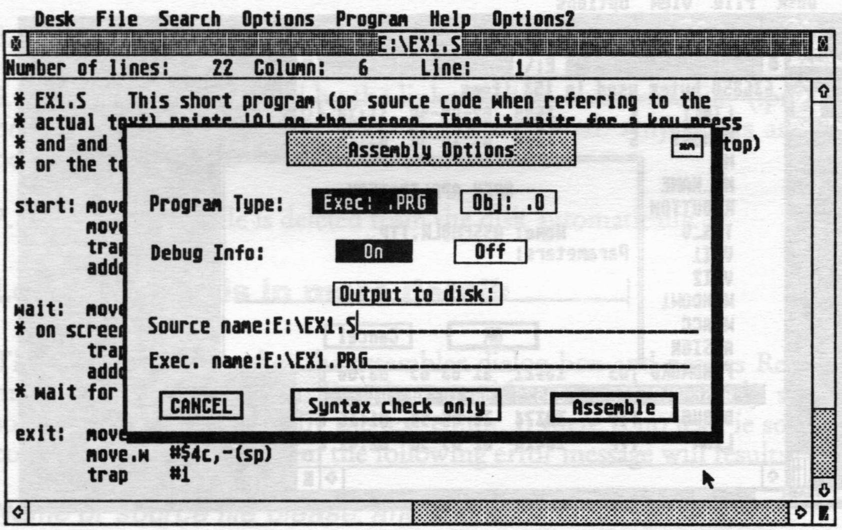

diagram 24:1 Assembler dialog box

A '.TTP' file is a special kind of executable program which expects parameters passed to it via a command line or via the 'p_exec' call. This type of program is often produced from C compilers as they produce this type of file almost by default. Double clicking on the this type of file results in a Tos Takes Parameters (TTP) dialog box being displayed – see diagram 24:2, and the parameters it needs are placed in the editable field – the command line. In this case the assembler would expect a file name with an extension '.s' (it must be in lower-case). You will only be presented by the TTP box if you double-click the assembler or linker from the desktop. If the file is on a different disk or in a different folder from that of the assembler then the path would have to be specified. The assembler and linker are both separately called using the 'p_exec' function from the editor. The source code file name and path is passed to the assembler and once it has been loaded it starts assembling the source code. If the source code file is on a different disk or in a different folder from the assembler and linker then the assembler's path must be set with *Set Paths...* from the editor prior to assembly.

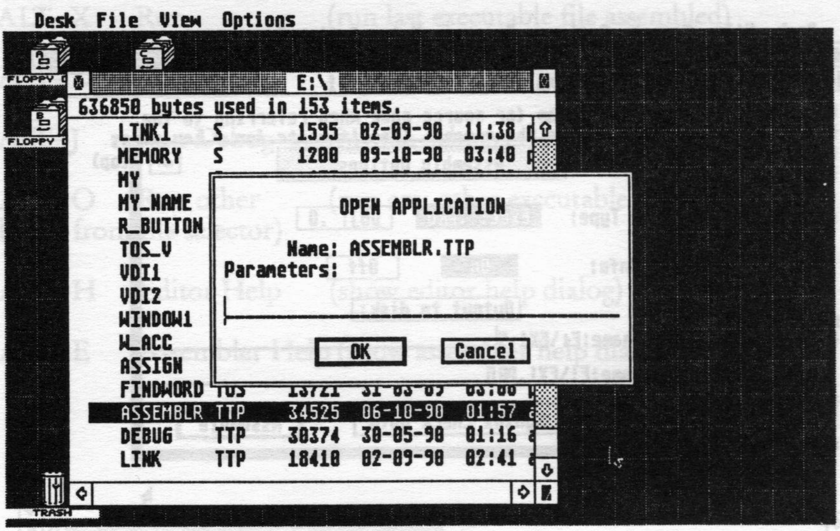

diagram 24:2 TTP dialog box

The actual process of producing an executable file from the source code held by the editor follows this pattern:

The programmer invokes the assembler dialog box and presses Return or clicks the 'Ok' button. Assembly then proceeds:

Assembly process:

1. The source code is automatically saved to disk.

2. The assembler is loaded using the 'p_exec' call. If the path is incorrect then the assembler cannot be loaded. Set path from *Set Paths...* drop down menu.

3. Source code parameters are passed to the assembler and the assembler launched.

4. The source code saved to disk is loaded by the assembler and analysed for errors. If there are any errors these are displayed on the screen and the assembly process aborted.

5. The assembler produces an object file if no errors occur.

6. The object file is loaded by the linker and an executable file (.prg) is produced and saved to disk if no errors are found. Any errors are reported to the screen.

7. The '.o' (object) file is deleted from the disk automatically.

Looking at this in more detail:

The programmer invokes the assembler dialog box and presses Return or clicks the 'Ok' button. For the assembler to work correctly some source code must be held by the text editor. If there is no text, ie source code held by the editor then the following error message will result:

Name of source file please, and name?

After the presence of text is found the source code held by the editor is automatically saved to disk, ie the whole of the text file being wholly or partially displayed in the editor window is saved to disk. The text is saved to disk on the path and with the name it was loaded with. This ensures that any changes made by the programmer to the source code are saved before the file is loaded by the assembler and analysed for mistakes. If the programmer has produced source code starting from scratch (ie no source code has been loaded) then the file name should be entered in the assembler dialog box. The path and name of the source code in the assembler dialog box 'Source name' field are taken from the original source path and name (when the file was loaded) and automatically placed there by the editor. The 'Exec. name' ie Executable file name field is automatically filled in by the text editor with the name and path taken from the source code name and path, but with the source code '.s' extension replaced by a '.prg'. The paths and files can be renamed.

Renaming the source code

If source code has been loaded from disk it is possible to change the path and name of the source code file being saved by the assembler so that the original file may be preserved if you wish. To do this the path and new name of the source code should be entered in the 'Source name:' editable field in the assembler dialog box. See diagram 24:1. In a similar manner the executable file name and path may be altered.

If source code has been written in the editor without loading a file from disk then the assembler cannot fill in the path and names of the source code and executable file. This must be done by yourself in the assembler dialog box. A typical path and name would be if assembling from floppy disk drive A:

Source name:a:\test.s

If the assembled program name was to be the same as 'test', ie 'test.prg' then it is sufficient to fill in the 'Source name:' field. The editor will provide the executable file name and path.

The assembler and linker path must also be set if they are held on a different drive or in a different folder from the original source code path. This should be done from the *Set Paths...* menu option. Once this has been set it will remain throughout until the editor is quit, or the path altered. See diagram 24:3

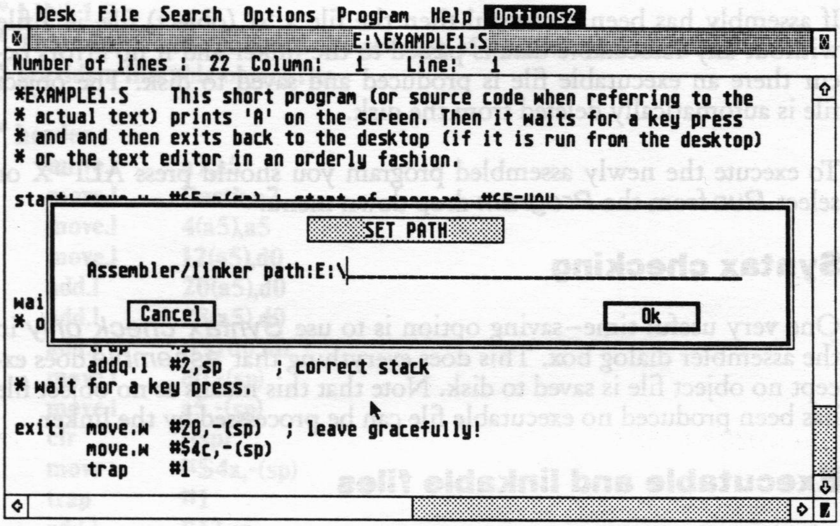

diagram 24:3 Set Paths dialog box

For the user who choses to keep the assembler, linker, text editor, all source code and executables in the root directory very little of the above need concern him.

After the source code has been saved then the assembler is loaded and launched into action. The assembler then loads the newly saved source code and analyses it for errors. Many types of error can occur but he main ones are usually easily corrected, as they are often simple typing mistakes or syntax errors.

To give an example:

```
	move.l	ao,d0
* etc
```

If errors occur during assembly then brief messages will be displayed on screen with the editor line number on which they occurred. These line numbers should be noted down so that they can be used in the text editor to rectify the errors.

If assembly has been successful then the file a '.o' (object) file, ie a file without any relocatable data is passed to the linker and if no errors occur there an executable file is produced and saved to disk. The object file is automatically deleted from the disk.

To execute the newly assembled program you should press ALT–X or select *Run* from the *Program* drop down menu.

Syntax checking

One very useful time–saving option is to use *Syntax check only* in the assembler dialog box. This does everything that *assemble* does except no object file is saved to disk. Note that this means as no object file has been produced no executable file can be processsed by the linker.

Executable and linkable files

At default the assembler is set up to produce '.prg' files ie GEM type executable files, and this can be seen in the assembler dialog box, 'Program Type:' selected radio button. The other option is to produce an object file. This is only useful if you wanted to link object files at a later time to produce an executable '.prg' file.

One of the main uses of producing linkable files is that once some particular source code has been assembled and debugged then it can be safely set aside until it is needed. For instance a GEM header file complete with stack and GEM arrays could be produced as a object file and then linked to whatever GEM program you have produced.

To demonstrate this procedure please examine the following source code shown below. LINK1.S is a GEM AES shell which provides some of the routines usually needed by any GEM program. This is first assembled with the 'Obj: .O' option selected in the assembler dialog box which produces a linkable object file. Next the file MY_NAME.S should be similarly assembled to produce an object file. Then they should be both linked using the *Link object files...* option from the *Program* menu. A suitable executable file name should be given to the linked files such as MY_NAME.PRG.

```
* LINK1.S
* This source code contains the essence of a GEM AES shell to
* provide a linkable object file

* header
	move.l	a7,a5
	move.l	#ustk,a7
	move.l	4(a5),a5
	move.l	12(a5),d0
	add.l	20(a5),d0
	add.l	28(a5),d0
	add.l	#$100,d0
	move.l	d0,-(sp)
	move.l	a5,-(sp)
	clr	-(sp)
	move	#$4a,-(sp)
	trap	#1
	add.l	#12,sp

* appl_intit()
	move.l	#appl_init,aespb
	jsr	aes	; call AES

	jsr	main

	.globl	exit
exit:
* appl_exit()
	move.l	#appl_exit,aespb
	bsr	aes	; call AES

	clr.w	-(sp)
	trap	#1

* AES subroutine
	.globl	aes
aes:	move.l	#aespb,d1
	move.l	#$c8,d0
	trap	#2
	rts
```

```
        ds.l    256
ustk:   ds.l    1

* GEM arrays
        .globl  contrl
contrl:         ds.w    12
        .globl  intin
intin:          ds.w    128
        .globl  intout
intout:         ds.w    128
        .globl  global
global:         ds.w    16
        .globl  addrin
addrin:         ds.w    128
        .globl  addrout
addrout:        ds.w    128

* some GEM functions
appl_init:      dc.w    10,0,1,0,0
appl_exit:      dc.w    19,0,1,0,0
        .globl  form_center
form_center:    dc.w    54,0,5,1,0
        .globl  aespb
aespb:  dc.l    contrl,global,intin,intout,addrin,addrout
        .globl  object_draw
object_draw:    dc.w    42,6,1,1,0
        .globl  form_do
form_do:        dc.w    50,1,2,1,0
```

Examining the above source code there are a couple of features that need looking at. One is the use of '.globl'. This makes this label and hence the address associated with it accessible to other files when they are linked. If they were not labelled .globl then any other reference to that label by another file would result in an error at link time. For instance the 'jsr main' instruction is made under the presumption that any file linked to it will have a label called 'main:' and that it will end with an 'rts'. If you look at the next program you will be able to see that 'main:' has been made '.globl'. This allows, at link time, the two files to be linked successfully and to operate correctly. Without 'main:' being '.globl' the link would fail with the warnings:

Undefined main from 'path\name__of__file'
Bad symbol type

Where 'path' would be something like a:\ and name__of__file could be 'link1'.

However, 'appl__init:' and 'appl__exit:' do not need to be declared globally as they are limited to use within the shell itself. Similarly 'ustk:'.

So it can be seen that only those labels that are used by other files than the one in which it is contained need to be declared '.globl', ie global.

Note also that linking is considerably faster than assembling which can be very useful when the program is a large project.

Note that the path name and/or the names of the object files and resultant executable file should be kept as short as possible. The reason for this is that a TTP program (the linker) is passed information about what files to link and the string that is used to pass this information can only be of a certain length. Going beyond this length results in the string being truncated. As one of the last parameters to be passed to the linker is the name of the executable file then this may be truncated. Suitable warnings are given to the programmer when using the linker.

However, it is possible to link many object files by passing a list of object files to the linker. To do this you have to leave the editor and use the linker from the desktop. Double click the linker and enter:

–f link.doc –o name.prg

This will load an ascii file from disk called 'link.doc' which should contain a list of object files that you want to link, and the resultant executable file will be called 'name.prg'. If the '.doc' file is on another disk or in another folder then you would need to do something like this:

–f b:\docs\link.doc –o name.prg
if 'link.doc' was on drive b:.

Similarly if you wanted the executable program saved to another disk or path, eg –o b:\name.prg

If you just enter:

–f link.doc

then the name of the first program in the 'link.doc' will be used as the name of the resultant executable file. So if 'xxx.o' was the name of the first object file in the list of object files then the executable file would be called 'xxx.prg'.

On the disk you will find a file named link.doc which you may like to experiment with.

The following program should be assembled to to make an object file and then linked with the above AES shell.

```
* MY_NAME.S
* This file provides the variable source code to be linked with
* the GEM AES shell.

        .globl main:
    move.l      #my_name,-(sp)
    move.w      #9,-(sp)        ; Gemdos function 'print a line'
    trap        #1
    addq.l      #6,sp           ; correct stack

* wait for key press
    move        #2,-(sp)        ; device number (console)
    move        #2,-(sp)        ; BIOS routine number
    trap        #13             ; Bios
    addq.l      #4,sp
    rts

my_name:        dc.b    "Roger Pearson",0
```

MY__NAME.S is a very simple file to link and it contains no GEM calls so we can try linking another more complicated file. The file below should be assembled and linked and given a suitable executable name such as DIAL.PRG. You will find DIAL.PRG on the disk as well as the object file.

```
* DIALOG1.S
* This example shows the construction of a simple dialog
* box by hand, how to diplay it on screen. (Taken from GEM4.S)
* This should be linked to LINK1.S

        .globl main
main:
    bsr         form_cent
    bsr         obdraw          ; put dialog box on screen
    bsr         f_do            ; handle interaction
    rts

form_cent:
    move.l      #form_center,aespb      ; get coords of centred tree
    move.l      #parent,addrin
    bsr         aes
    movem.w     intout+2,d0-d3          ; returned in intout+2
    rts

obdraw:
    move        #0,intin        ; index of first object
    move        #1,intin+2      ; depth
    move        d0,intin+4      ; x coord
    move        d1,intin+6      ; y coord
    move        d2,intin+8      ; width
    move        d3,intin+10     ; height
    move.l      #parent,addrin ; address of parent dialog box tree
    move.l      #object_draw,aespb
    bsr         aes
    rts

f_do:  move.l   #form_do,aespb
    clr.w       intin           ; No editable text field
    move.l      #parent,addrin
    bsr         aes
    rts

text1:  dc.b    '  -----EXAMPLE----',0

text2:  dc.l    texty,textt2,textt2
```

```
        dc.w    3,0,2,$11f0,0,3,5,0

texty:  dc.b    'Exit',0

textt2: dc.b    0

* dialog box tree
parent:
        dc.w        -1,1,2,20,0,16          ; g_box
        dc.l        $00021100
        dc.w        170,100,250,100

        dc.w        2,-1,-1,28,0,0          ; g_string, title string
        dc.l        text1
        dc.w        10,10,5,1

        dc.w        0,-1,-1,22,7+32,0       ; g_boxtext, boxed exit button
        dc.l        text2
        dc.w        50,60,60,25
```

At default 'Debug Info:' is switched on but unless the '.globl' statement is used to make the label global having debug switched on will have no effect. However, switching this off will result in no debugging information being passed to the executable file despite any labels being declared '.globl'.

By debugging information we mean that any labels that are declared global can be used in the debugger for reference. As it is also possible for other programmers to use this information in a debugger, and it makes the program length larger, it is usual to leave this information out in the final executable file.

The Debugger

In chapter three some ground was covered using the disassembler although this was rather brief. The debugger has an accompanying document on disk which refers to the 'szadb' program. If you read the documentation you will soon realise that 'adb' is a UNIX type debugger, and that 'szadb' has been based upon it.

The document is not what could be termed user friendly! However, one of the prime reasons of using a debugger is to either single–step a program to a certain point and the other is to run the program at full speed until that point (usually to find a bug, by examining what happens when single–stepping) and doing this using the commands as listed is not too difficult.

As discussed in chapter three the command for single–stepping is ':s' and for running at full speed is ':c'. In both cases using upper–case letters results in all the registers being displayed on screen. The command for setting a breakpoint is ':b', and for deleting a breakpoint is ':d'.

A breakpoint is used to stop a running program at a particular point. In the course of programming it is usual to come across bugs, ie the program does not do what we expect! The result of this is the ST locking up or bombs appearing on screen or peculiar behaviour, etc. As it is usual to write programs of any length as modules consisting of subroutines then it is usually possible to have some idea where the fault may lie.

For instance when writing a word–processor it may follow the following path:

Do GEM header set user stack
Open GEM application
Load Resource file
Find dialog/menu addresses
Open menubar
Declare type and size of GEM window
Open a GEM window
Fill window with white background
Put cursor on screen
Wait on event_multi etc

Now in the course of writing this imaginary word–processor we would probably pause every so often and assemble what we had written and test the results of our endeavours by running the resultant executable file. Now in the course of running the program if a fault occurred say for instance just before the GEM window was opened then we would expect that the fault would lie before this point. If we examine the source code and check that everything looks ok and we cannot fault what we have written it is time for the debugger!

We could then declare a label global in the source code after the point where the menu had been placed on screen. We could then assemble the source again, enter the debugger set a breakpoint at the label, and run the program at full speed until this point. The program would halt at this point, hopefully before the bug occurs and then we could single–step each line of code until the fault occurs. It is not usual to single–step a trap as the code is often extensive and is not alterable in any case, as it is in ROM. We also have to assume that it is correct. By keeping an eye on the registers and parameters we can usually find the fault, though sometimes not for after a while of searching.

Looking at the practical details of doing this:

If we called the label 'fault:' prior to the routine the we believed was causing the bug, then setting a breakpoint after we enter the debugger would be done like this:

fault :b (press Return key)

Once this has been done then we should run the program at full speed by using ':C'. The program will halt at the breakpoint and we can now single–step the faulty routine until the bug occurs.

What about avoiding single–stepping 'traps'?

When we see that a trap is about to be executed as we are single–stepping we should immediately set a breakpoint by the command ':b'. Then the program should be run with ':c' and after the trap has been executed ':s' should be used to return to single–stepping. Pressing return operates the last command again, so unless we use ':s' after the trap has executed pressing Return will run the program until the end or until a fault occurs.

What about subroutines and dbra's?

In the course of debugging we often come across a subroutine (jsr) that we know is ok as we have used it before with no unpleasant results, or we come across a 'dbra' routine that we know is ok too. How can we avoid having to single–step these routines, if a label is not available after the routine so that we can set a breakpoint there?

We can use the './i' command which will disassemble each instruction from the program counter, ie from where we are in the program and at the same time will give us the address of the instruction either as an offset of a label or an actual address. Pressing Return will disassemble the next instruction and so on. Once we arrive at a point at the end of a subroutine or after a dbra instruction then we can set the breakpoint there as detailed above. For instance the address given after a dbra might be 'fault+32', then we should use 'fault+32 :b'. Use '&' to return to original spot (prior to using './i') in the debugger.

To delete a breakpoint we should use ':d'. So to delete the breakpoint after the dbra then we should use 'fault+32 :d' and press Return.

Using '$b' will list all the breakpoints if any. ':d' will delete one of the listed breakpoints. So to delete a list of breakpoints '$b' should be followed by a ':d', and this procedure should be followed until all '$b' shows no breakpoints.

'$e' will list all global symbols and their addresses.

We sometimes need to find the contents of a symbol, for instance the state of some particular flag, or symbol that has stored an address. Often we need to see whether a symbol contains what we think it should. For instance we may expect the label 'screen_address' to contain the screen's address but what if inadvertently it has been over-written by some stray code. Note that 'screen_address' would have been declared to have a value of a long word. If at the start of the program we note the value placed in the symbol then if we later inspect it at another point in the program we can confirm that it holds the correct value.

To examine any particular symbol we should use 'symbol/X'. For instance if examining the 'screen_address' label we should first use '$e' to see the actual length of the symbol. It would be truncated to 'screen_a:', ie eight characters, excluding the colon, which is the maximum allowed. So to see the value it contained we would write 'screen_a/X', were 'X' indicates a long word value is to be returned. This would give us the address we want, hopefully. To get a word value we would use 'x'. See 'FORMATS' in the 'szadb' documentation.

In the case where we expected a string we would use an 's'. For instance 'my_name/s' would give 'Roger Pearson' if we were to examine the first example programs in this book in the debugger.

Use '$q' to leave the debugger at any point. Note though that leaving a GEM program in the middle of it may result in faults appearing later on in the editor and/or the next time we want to use the debugger. This may because we have not closed, or deleted a window, used 'appl_exit', not released a resource file space, etc. It is better to quit a GEM program correctly if at all possible so that all memory, windows, menus, resources can be dealt with properly by your exit routine.

Reserved words

As you might expect the assembler will get very confused if you use labels such as 'a0' or '.bss' to refer to anything other than what the assembler thinks they are, namely register a0, and the block storage segment. The following list shows what words are reserved, ie they are understood by the assembler to refer to a specific function.

.bss, .comm, .data, .dc, .ds, .end, .equ, .even, .globl, .org, .text; and similar words all without the preceding period.

a0 to a7; d0 to d7; pc, sp, sr, usp

Error messages

The assembler and linker will report errors whenever it finds one as it goes about assembling and linking your program. These errors are often caused by bad syntax, and spelling mistakes. The following list gives the most common errors that are likely to occur in the normal course of assembling and linking.

Error message	Example fault	Solution
Syntax error	:main:	main:
Missing ':' after label	movvve	move
Illegal expression	move.l #$name,-(sp)	move.l #name,-(sp)
Illegal use of symbol in expression	move.l #name,-((s.p)	move.l #name,-(sp)

Missing ')'	–(sp	–(sp)
Invalid decimal constant	move #9b,–(sp)	move #9,–(sp)
Illegal instruction	trap #19	trap #1
Non-terminated string	dc.b "roger	dc.b "Roger",0
Bytes not separately relocatable	dc.b "roger",p	dc.b "Roger",0
Missing ':' after label	RTS	rts

The last error message will appear when ever upper–case text is used.

the first example programs in this book in the debugger.

Use '$q' to leave the debugger as appropriate. Note though that leaving a GEM program in the middle of it may result in faults appearing later on in the editor and/or the next time we want to use the debugger. This may because we have not closed, or deleted a window [illegible] not released a resource file space, etc. It is better to quit a GEM program correctly if at all possible so that all [illegible] resources can be dealt with properly by your exit routine.

Reserved words

As you might expect the assembler will get [illegible] you use labels such as 'a0' or '.bss' to refer to anything other than what the assembler [illegible]. The following list shows what words are reserved, ie they are understood by the assembler to refer to a specific function.

.bss, .comm, .data, .dc, .ds, .end, .equ, .even, .globl, .org, .text; and similar words all without the preceding period.

a0 to a7; d0 to d7; pc, sp, sr, usp

Error messages

The assembler and linker will report errors whenever it finds one as it goes about assembling and linking your program. These errors are often caused by bad syntax, and spelling mistakes. The following list gives the most common errors that are likely to occur in the normal course of assembling and linking.

Error message	Example fault	Solution
Syntax error	:main:	main:
Missing ':' after label	movve	move
Illegal expression	move.l #$name,–(sp)	move.l #name,–(sp)
Illegal use of symbol in expression	move.l #name,–((s.p)	move.l #name,–(sp)

Appendix
Key Codes

This appendix list all the keycodes available in hexadecimal. The first two numbers refer to the actual key struck, whilst the last two numbers refer to the ASCII code.

```
* KEY.S
* get keycode as result of keypress in register d0, from BIOS bconin()
* eg character 'a'=001e0061. ASCII character returned in least
* significant byte, and scancode (physical key code) is returned
* in least significant byte of high word.
* GEMDOS 'cconin' can also be used.
* 'evnt_multi' returns keycodes as listed.

* bconin() read a character
        move        #2,-(sp)        ; device number (console)
        move        #2,-(sp)        ; BIOS routine number
        trap        #13             ; Call Bios
        addq.l      #4,sp
* d0 has keycode

exit:
        move.w      #20,-(sp)  ; leave gracefully!
        move.w      #$4c,-(sp)
        trap        #1
* exit from program properly
```

Main Keyboard

Unshifted			Shift	Control	Alt
a	1e61	A	1e41	1e01	1e00
b	3062	B	3032	3002	3000
c	2e63	C	2e43	2e03	2e00
d	2064	D	2044	2004	2000
e	1265	E	1245	1205	1200

f	2166	F	2146	2106	2100
g	2267	G	2247	2207	2200
h	2368	H	2348	2308	2300
i	1769	I	1749	1709	1700
j	246a	J	244a	240a	2400
k	256b	K	254b	250b	2500
l	266c	L	264c	260c	2600
m	326d	M	324d	230d	3200
n	316e	N	314e	310e	3100
o	186f	O	184f	180f	1800
p	1970	P	1950	1910	1900
q	1071	Q	1051	1011	1000
r	1372	R	1352	1312	1300
s	1f73	S	1f53	1f13	1f00
t	1474	T	1454	1414	1400
u	1675	U	1655	1615	1600
v	2f76	V	2f56	2f16	2f00
w	1177	W	1157	1117	1100
x	2d78	X	2d58	2d18	2d00
y	1579	Y	1559	1519	1500
z	2c7a	Z	2c5a	2c1a	2c00
1	0231	!	0221	0211	7800
2	0332	"	0322	0300	7900
3	0433	£	049C	0413	7a00
4	0534	$	0524	0514	7b00
5	0635	%	0625	0615	7c00
6	0736	^	075e	071e	7d00
7	0837	&	0826	0817	7e00
8	0938	*	092a	0918	7f00
9	0a39	(	0a28	0a19	8000
0	0b30	)	0b29	0b10	8100
–	0c2d	__	0c5f	0c1f	8200
=	0d3d	+	0d2b	0d1d	8300
‘	2960	-	29ff	2900	2960
\	2b5c	|	2b7c	2b1c	2b5c
[	1a5b	{	1a7b	1a1b	1a5b
]	1b5d	}	1b7d	1b1d	1b5d
;	273b	:	273a	271b	273b
’	2827	@	2840	2807	2827

,	332c	<	333c	330c	332c
.	342e	>	343e	340e	342e
/	352f	?	353f	350f	352f
Space	3920		3920	3900	3920
Esc	011b		011b	011b	011b
Backspace	0e08		0e08	0e08	0e08
Delete	537f		537f	531f	537f
Return	1c0d		1c0d	1c0a	1c0d
Tab	0f09		0f09	0f09	0f09

Cursor Pad

Unshifted		Shift	Control	Alt
Help	6200	6200	6200	Print screen
Undo	6100	6100	6100	6100
Insert	5200	5230	5200	left button
Clr/Home	4700	4737	7700	right button
Up–arrow	4800	4838	4800	move mouse up
Dn–arrow	5000	5032	5000	move mouse down
Rt–arrow	4b00	4b34	7300	move mouse right
Lft–arrow	4d00	4d36	7400	move mouse left

Numeric Pad

Unshifted		Shift	Control	Alt
(	6328	6328	6308	6328
)	6429	6429	6409	6429
/	652f	652f	650f	652f
*	662a	662a	660a	662a
–	4a2d	4a2d	4a1f	4a2d
+	4e2b	4e2b	4e0b	4e2b
.	712e	712e	710e	712e
Enter	720d	720d	720a	720d
0	7030	7030	7010	7030

1	6d31	6d31	6d11	6d31
2	6e32	6e32	6e00	7e32
3	6f33	6f33	6f13	6f33
4	6a34	6a34	6a14	6a34
5	6b35	6b35	651b	6b35
6	6c36	6c36	6c1e	6c36
7	6737	6737	6717	6737
8	6838	6838	6818	6838
9	6939	6939	6919	6939

Function Keys

Unshifted		Shift	Control	Alt
F1	3b00	5400	3b00	3b00
F2	3c00	5500	3c00	3c00
F3	3d00	5600	3d00	3d00
F4	3e00	5700	3e00	3e00
F5	3f00	5800	3f00	3f00
F6	4000	5900	4000	4000
F7	4100	5a00	4100	4100
F8	4200	5b00	4200	4200
F9	4300	5c00	4300	4300
f10	4400	5d00	4400	4400

Bibliography

Compute!'s Technical Reference Guide series: all by Sheldon Leeman. These are recommended guides to the ST and contain many examples of source code in C, and assembler.

Volume 1: AES
Volume 2: VDI
Volume 3: TOS

Concise Atari 68000 Programmer's Reference, Katherine Peel, published by Glentop.

Programmer's Guide to GEM by Balma and Fitler, published by Sybex.

Various assembly and C source code, programming information text files, example programs, etc, available from PD libraries:

Goodman Enterprises **0782 335650**
16 Conrad Close
Meir Hay Estate
Longton
Stoke–on–Trent
ST3 1SW

Softville **0705 266509**
Unit 5
Stratfield Park
Elettra Avenue
Waterlooville
Hants
PO7 7XN

MT Software **0983 756056**
Greens Ward House
The Broadway

Totland
Isle of Wight
PO39 0BX

ST Club **0602 410241**
49 Stoney St
Nottingham
NG1 1XF

Recommended assembly language development package, necessary for programs of any complexity, and size:

DEVPAC 2, from HiSoft.

Recommended BASIC:

GFA BASIC Version 3

Recommended C language:

Sozobon C– an excellent PD offering.

Lattice C Version 5, from HiSoft

Glossary

A small glossary of some programming terms

680x0 16/32 bit CPU manufactured by Motorola. Used by ATARI ST, CBM Amiga, Apple Mac.

Active Window The GEM window which is on top of any other windows and thus receives all mouse, and keyboard input/output.

.ACC GEM desk accessory executable file.

Address A number that identifies a particular location in the computers memory – RAM or ROM.

Address Register 32 bit register used to store addresses, numbered a0–a7.

AES Application Environment Services: part of GEM that provides windows, forms and menus.

Alert A standardized dialog box which contains a short message, and usually a NOTE, WAIT, or STOP icon.

Alogorithm Precise sequence of steps required to perform some action.

.APP GEM executable file. APP=APPlication

Application A computer program that performs something useful, eg word processor, spreadsheet, compiler.

Array A structure for storing data in sequential locations/order. Sometimes known as a buffer.

ASCII American Standard Code for Information Interchange. Used to represent alphanumeric characters held as bit images in memory.

Assembler A program which translates source code to object code.

Assembly Language A low level computer language.

ASSIGN.SYS ASCII file used by the VDI to configure the system, usually with particular fonts.

BBS **B**ulletin **B**oard **S**ystem. A program running on a remote computer which handles communication between your computer and the other computer. Communication is via modem.

Binary A system of numbering using 0, and 1. Base 2.

Bit **B**inary di**IT**, which can have a value of 1 or 0.

BITBLIT Bit image block transfer.

Bit map A collection of pixels used to represent an image.

Boolean logic Operations performed on binary numbers.

Bug An error in a program.

Button An outlined area (rectangular box) in a dialog box that you click in to do something. The mouse button usually refers to the left mouse button.

Byte 8 bits. The standard length of a location in memory.

CCR Condition Code Register. Bits (called flags) in the CCR indicate the results of a program operation.

Click The user positions the mouse pointer and presses a mouse button and releases it.

Click-drag The user positions the mouse pointer presses a button and without releasing the button moves to another location on the screen. The button is then released.

Clipping Rectangle A rectangle that defines the bounds of VDI graphics display. Any graphics drawn outside the clip rectangle will not be drawn.

Compiler A program that converts high level language source code to object code.

CPU Central Processing Unit, eg 68000

Crash When the program stops working properly. Usually followed by bombs, or a hang or both.

Data Register Registers d0–d7, which can hold data up to 32 bits long.

Debugger A program used to locate bugs in a program. Also known as a monitor, or disassembler.

Default What you get if you do not specify something different.

Desk Accessory An application that must be in the root directory at boot up, on drive A, or partition C if hard disk. Accessible from the left–most drop down menu.

Desktop GEM user interface that compares the default display with an office desk.

DESKTOP.INF ASCII file which contains the users desktop preferences, such as whether a window should be opened at boot up, resolution, etc.

Dialog A form usually designed with a resource construction kit, that displays information and often allows the user to input information.

Directory The contents of a disk usually displayed in a GEM window.

DOS **D**isk **O**perating **S**ystem, mostly associated with MSDOS on the IBM PC and compatibles.

Double-click Pressing the mouse button quickly twice in succession usually to activate an object or icon.

DR **D**igital **R**esearch– the creators of GEM.

Drop-down menu Usually located at top of screen and activated by moving mouse pointer there.

Field A specific part of a form that is usually user editable. For instance in a mailing record there would be a name field, an address field etc.

Font A collection of letters, numbers, and symbols with a consistent look.

Full box A GEM window function which allows the user to expand a window to its full size or return it to its previous size.

Garbage Meaningless or unexpected characters.

GDOS **G**raphics **D**evice **O**perating **S**ystem. Usually auto-booted so that fonts can be loaded and printed. Used by art and DTP programs.

.GEM A metafile which contains a list of VDI operations, often used by paint programs.

Greyed Text that is fainter than its usual look. Often indicates something that is not selectable.

Hang The application cannot accept any input or proceed. ST has to be reset to be able to continue.

K A measure of a computers memory, disk space that is equal to 1024 characters, or bytes.

Invoke Run or launch a program or application.

MC **M**otorola **C**orporation, manufacturer of 68000 CPU.

Meg, megabyte A measure of a computers memory, disk space. Equal to 1024K. or 1,048,576 characters or bytes.

Menu bar The horizontal area across the top of the screen that holds the menu titles.

Object code The result of assembling source code.

Operating System A program which controls the day–to–day running of the computer.

Program A piece of software that usually does something useful. Other names for program: process, application, routine.

Patch A small program used to correct or enhance another program.

RAM **R**andom **A**ccess **M**emory. Used for the short term retention of memory until the ST is reset or turned off.

RGB **R**ed **G**reen **B**lue

ROM **R**ead **O**nly **M**emory. Memory that can be read but not written to. O/s is stored in ROM.

Root directory The directory that appears when a disk icon is double–clicked or opened.

Source code The text of a program that can be read/edited by the programmer.

ST **S**ixteen **T**hirty two. Named like this as the ST has an internal 32 bit bus, and an external 16 bit bus.

SySop A BBS's operator (Sysop= System operator)

String A sequence of characters; a word, phrase, or number.

Wildcard A symbol (usually *) that means any character or sequence of characters. In the file selection box '*.DOC' would display all files ending in DOC.

Index

O

P

Q

R

S

T

U

V

W

Introducing ATARI ST machine code

At last a book (and disk) that takes the beginner step–by–step into the secrets of programming the ST in its native language: 68000 assembly language. Now for the first time zzSoft provide a complete programming environment for the beginner to intermediate ST applications programmer with a book and a disk. No need to buy a separate assembler or resource kit!

For any ATARI ST, STE, Mega: 520, 1040 or more

The book has over 40 complete assembly language examples, with an index, and glossary.

24 chapters introduce the gamut of ST applications programming from formatting a disk to constructing drop down menus and dialog boxes.

The book comes complete with a disk which has:

- A complete symbolic 68000 assembler and linker.
- A resource construction kit, for the easy creation of drop down menus, and dialog boxes.
- A symbolic debugger.
- An integrated fully featured GEM text editor specially written for this book – assemble the source code and run the executable program from within the editor!

All the source code in the book is on the disk and is ready for assembling.

The disk also contains a comprehensive listing of all BIOS, XBIOS, GEMDOS, AES and VDI assembly language calls, with examples.

zzSoft

ISBN 1 873423 01 2