Modern Atari System Software

A Programmer's Guide

Another Quality Reference Guide for your Atari 680x0 Computer from

HiSoft
High Quality Software

Modern Atari System Software
An Introduction

By HiSoft

Book:

written and compiled by Tony Racine and Alex Kiernan.

Published by HiSoft

The Old School, Greenfield, Bedford MK45 5DE UK

First Edition, July 1993 - ISBN 0 948517 63 8

Table of Contents

Preface

With the introduction of a new computer from Atari, a new multitasking operating system and a radically new replacement for GDOS, much has changed! This book does not try to be an authoritative guide to all aspects of the new machine and the updated system software, but it does contain an outline discussion about many aspects of the improvements, especially when extra background information is required to help document the calls printed here.

Whilst this book show shows calling conventions for the functions based around the HiSoft products HiSoft BASIC 2, Devpac 3 and Lattice C 5, Atari programmers who are not using one of these products should still find the information contained within this book useful in conjunction with a language using the standard Atari binding names. At the time of writing much of this material was not generally available and to our knowledge is printed here for the first time.

For those users using one of the above HiSoft language products the *Language Specific Issues* appendix includes information on how the calls discussed here are made available to a program.

Credits

HiSoft would like to acknowledge the use of material from a variety of sources in the compilation of this book. Extensive use was made of documentation produced by Eric Smith in the MiNT section and other Atari information. Other (unwitting) contributors to this guide were Bill Rehbock and the member of Atari's TOS group (USA) and David Nutkins of HiSoft.

Chapter 1
The Falcon030

The Video sub-system

The new Falcon video hardware supports graphics modes which allow some older ST software to run without modification, however, programs which are written to run regardless of the given screen resolution or colour palette size, should be able to take advantage of some quite spectacular improvements over the older machines.

In terms of the screen sizes, the actual hardware within the machine would appear to be much more powerful than the new operating system calls will allow, however at present the maximum screen size which can be obtained on either a VGA style monitor or TV are 768 pixels across the screen in overscan mode and 480 lines down. The screen resolution is fairly programmable and certain options exist within the operating system to pick and choose certain combinations of features.

The colour modes of the computer can be selected by determining the number of bits dedicated to representing the colours. At present there are 5 colour depths: 2, 4, 16, 256 and 65536 (or 32768) colour modes. The first 4 modes feature user programmable palettes, the last mode is known as *true colour* mode; that is to say that the values placed into memory literally represent the colours displayed on the screen where as the former modes only represent a position in a colour look up table.

Software Video Mode Selection

The Falcon OS calls support the old screen and colour resolutions used on the previous ST computers wherever possible. Historically the ST had only 3 screen modes, low resolution colour, medium resolution colour and high resolution monochrome; these were augmented by an additional set of modes on the TT (note that the TT video XBIOS calls are not supported on Falcon). An old screen call would simply require a single digit parameter to determine which screen was currently in use or due to be set.

The complexity of the new Falcon hardware means that such a simple system is now quite impractical, so the new operating system calls can accept the old ST style `rez` code for ST compatibility only, but a more sophisticated `modecode` is now used to describe the attributes which are desired to be set up on the screen for non-ST resolutions.

The `modecode` takes the form of a 16 bit word, individual bits of which are used to select the status of different screen functions. The top seven bits of the high word are reserved for future expansion, the remaining 9 bits take the form:

Hi byte	Low byte
X X X X X X X F	S O P V 8 N N N

The bits have the following meanings:

X	Reserved for future expansion
F	Vertical flag.(Line double/Interlace)
S	ST compatibility flag (for ST low, medium & high res)
O	Overscan flag. When set, it multiplies the screen pixel width and height by 1.2. This should not be set for use on a VGA monitor.
P	PAL flag. PAL mode when set, NTSC when clear.
V	VGA flag. VGA mode when set, TV mode when clear.
8	80 column flag. 80 column mode when set, 40 column when clear.
N	Number of Bits Per Pixel (This determines the number of on-screen colours). 0 = 1 BPP, 1 = 2 BPP, 2 = 4 BPP, 3 = 8 BPP, 4 = 16 BPP, or 2, 4, 16, 256 or 65536 colours respectively. The 16 bits per pixel mode is also known as true colour mode.

Please note that a couple of `modecode` permutations are not allowed, these are: 40 column 2 colour mode on TV or VGA and 80 column true colour on a VGA monitor.

When using the operating system calls to set the video modes, please note that the calls do not check the validity of the code and it is quite possible to attempt to set modes which are either impossible to generate by the hardware or to display on the current monitor or TV. It is left to the user to enquire about the display device and to select the correct modes accordingly.

The Audio sub-system

The Falcon sound system is massively improved over the older ST range. A whole new section of the computer is dedicated to the ability to play multi-track stereo sampled sound as opposed to the single chip sound generator which the Falcon retains for machine compatibility. All this in conjunction with the on-board DSP (more on this later), combines to produce what is probably the most potent sound system currently available on any home computer today.

For example, the sampling section of the computer features on-board 16-bit stereo sampling device as standard but this can be extended by the addition of external circuitry to facilitate a full 4 stereo tracks of sound input and output. An input/output switch matrix located within the Falcon's circuitry can connect the input of the Digital to Analogue converters (DACs) to the computer's DMA system as it can the output of the Analogue to Digital Converters (ADCs). This permits direct to disk recording and playback as a standard feature of the machine which on its own will place Atari once again at the top of the professional recording musicians pedestal, an act which has always been difficult to follow in any case.

In component terms, the sound system features a G.I./Yamaha sound generator chip (a feature of the older ST computers), a 16-bit stereo CODEC chip and a digital input/output switch matrix controller which has full access to the internal DMA structure of the computer and externally connectable clock source. Another feature of the computer is the on-board DSP, a complete sub-system in its own right, which can be used as a stand alone processor within the case of the Falcon, however, since the DSP and the CODEC can be connected together, the two devices working with each other offer formidable possibilities in sound generation and processing. The rest of this section will try to outline each of the component parts of the sound system.

The CODEC

The CODEC (a contraction of enCOder-DECoder) is a 16-bit stereo analogue to digital and digital to analogue converter all in one compact chip. The device features sixty four times over-sampling and digital filtering of the analogue signals flowing both into and out of the device.

All of this adds up to a very high quality audio device which is built into the computer as a standard item. The CODEC is provided with a high speed clock with the sampling speed is governed by both this clock, and a series of sub-dividers located within the Falcon circuitry.

The Falcon hardware also allows an external clock source to be provided. This means that other sampling rates which are not otherwise possible with the standard machine can be produced very easily, most notably the recording industry rates of 44.1kHz (for CD) and 48kHz (for DAT).

External Expansion

The DSP port, mounted on the rear of the computer, enables the user to add a variety of peripherals to the Falcon via a high speed serial interface. Devices such as fax modems, low cost laser printers and scanners could all be connected to the machine via this socket. From the sound system's point of view probably the most interesting aspect of this socket is that it allows the expansion of the internal 16 bit stereo sound facilities by a further 3 stereo channels to 4 tracks of stereo sound or possibly 8 tracks of mono sound; provision for the control of this hardware is made in the operating system calls.

The inputs and outputs provided on the DSP port are routed through the digital switch matrix so the signals can be redirected to any of the system transmitters or receivers.

The switch matrix

This seemingly simple device controls the routing of inputs and outputs of the computers DMA, CODEC, DSP and external high speed serial ports with one another. For example, it is possible to connect the high speed external serial input to the DMA record channel to facilitate the input of data into the Falcon from a device such as a scanner. By re-routing the data input by the switch matrix, it is possible to direct that input to the DSP instead, where upon the DSP may be requested to perform some sort of image enhancement or data compression before passing it on to the Falcon's internal memory, such is the flexibility of this I/O system.

All data is transferred around the system using a high speed serial data stream, capable of speeds of up to 8 million bits of data per second. The user has the option of selecting a continuous or non-continuous data mode. In non-continuous mode, the transmitter and receiver implement a full 'sender-receiver' handshaking system, this prevents data overrun and underrun within the system. Data which is transferred using this mode is guaranteed to be picked up at the receiving end without loss, since the receiver will not request the next piece of data until it has acknowledged acceptance of the first. The non-continuous mode of transfer is recommended for use when ever an application does not need to move a maximum amount of data in a minimum or fixed time slot.

By turning off the handshaking transfer mode, it is possible to transfer data between any two given devices at very high speeds indeed, in theory, rates of up to 1 Mbyte (8 Mbits) of data per second can be achieved. The continuous data mode should only be used when a high speed of data is necessary and that the application can be sure that no other bus-intensive hardware within the machine is going to be used. Since the DMA record and play channels are connected to the Falcon address and data buses, any other bus intensive activity, such as running the video screen in true colour mode for example, may prevent the DMA FIFO (First In, First Out) buffer from being filled or cleared before the next transmission or reception time frame. If the 32 byte buffers are not completely filled or read out before the next transmission begins, then either an underrun or overrun situation will occur.

At this point it may be worth noting that the continuous mode of transmission is probably the most efficient way of feeding the DSP with data, since the DSP interrupt structure does not seriously affect its processing speed and the device is quite possibly sitting around waiting for a whole block of data to be received before it can get on with its processing task.

The DSP sub-system

Although this part of the Falcon may correctly be considered to be part of the Falcon's sound system, the truth is that it can be used totally independently of the sound and is a very useful feature for other applications in its own right. As may become clear, the DSP is a significant enough feature of the Falcon that it deserves a very close inspection.

What is a DSP?

The letters DSP together form an acronym which stand for Digital Signal Processor. In essence, this is a form of high speed, single chip microprocessor that is specifically designed to perform very high speed digital data processing.

Much is made of the Falcon containing the powerful Motorola 68030 microprocessor but with the Motorola 56001 DSP, the Falcon actually contains two quite separate processors. The Falcon DSP is a special chip which is in some ways similar to the 68030 in that it has its own RAM, it can support its own ROM (not present on the Falcon) and it can load and run programs and receive, process and output data. They are similar to the extent that even some of the programming instructions are the same!

The DSP56001 architecture

Internally the 68030 and 56001 are very different; whilst a detailed description of the DSP56001 is beyond the scope of this manual, other books are available on this subject anyway, but a brief outline of this device follows.

Where as a microprocessor is designed to be easy to program and perform a number of different tasks, everything from word processing and desktop publishing to controlling washing machines and factory machining equipment, the newer Digital Signal Processing chips, as their name implies, are designed to accept high speed digital data (such as from an analogue to digital converter), process it in some way and then output the digital result (usually to a digital to analogue converter). Since the device is designed for maximum data throughput, many aspects of the way in which the device has been designed represent a no compromise solution and are not necessarily the most straight forward or intuitive - just plain fast!

As a result, the DSP architecture may leave a microprocessor programmer cold at a first and wondering 'What the heck did they do that for?'. The DSP really is a very efficient number cruncher indeed. For example the 68030 would take about 28 clock cycles to perform a 16 bit by 16 bit signed multiply and produce a 32 bit result. With a Falcon running at 16 MHz, this will take about 1.6 micro seconds, that's about 571,000 multiplies per second - not bad. However the Falcon's DSP runs at 32 MHz and will take two clock cycles to perform a 24 bit by 24 bit multiply yielding a 48 bit result. This will therefore take 62.5 nano seconds to perform which is about 16 million multiplies per second - Phew! OK, so the chances of running flat out at this speed are slim, but the DSP really is very rapid at this kind of work since most instructions are capable of executing in 2 or 4 clock cycles (1 or 2 instruction cycles).

In actual fact, the reason for the slightly odd design of the device is that it will enable the DSP to do several things in parallel, so it can actually be much faster than we have already demonstrated, the trick is knowing what operations the DSP can do in parallel to know how to get the most out of it. For example, to take our previous demonstration one step further, the 56001 is actually quoted as being able to perform a 24 bit by 24 bit multiply, with a 56 bit addition onto the result, two data moves, two address pointer updates and an instruction pre-fetch all in 2 clock cycles (or 1 internal instruction cycle).

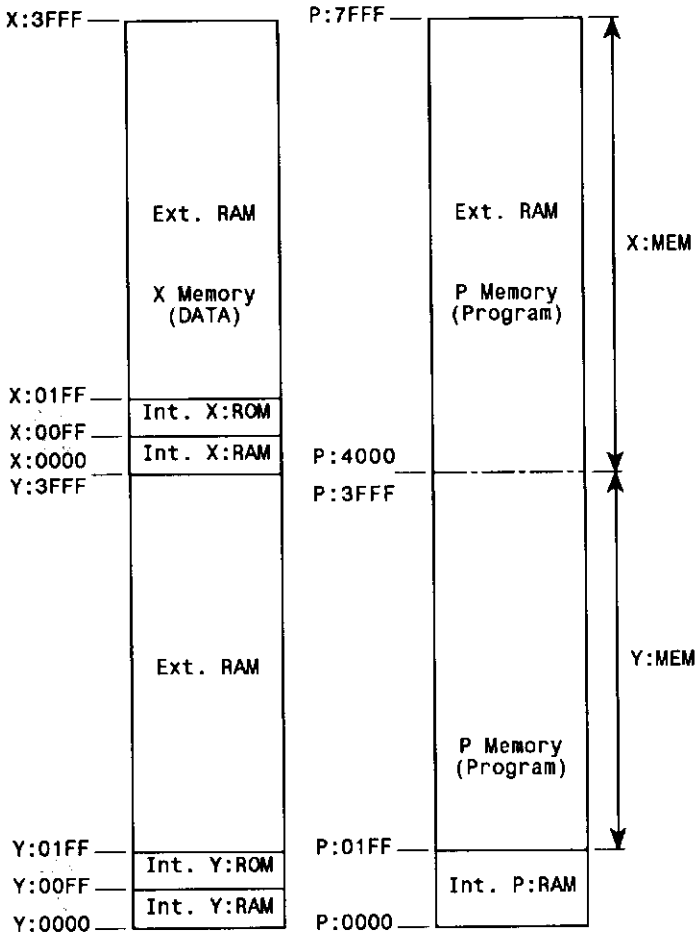
Part of the key to this staggering performance is in the fact that the processor does not support one address bus (like the 68030), but notionally three! These are known as the P memory (Program) and the X and Y data memories. This type of configuration is known as a Harvard architecture, after the university in America that pioneered this type of technology. This form of addressing is fast because it contains three separate sections of circuitry which can each pick up data and instructions simultaneously and process that data in a couple of cycles. In contrast a more conventional device, such as the 68000 processor, would need to pick up the next instruction to be executed followed by the one or two lumps of data to be processed, one after the other. This sequential series of memory accesses clearly must be slower than the parallel technique.

Since the DSP is designed as a fast mathematical processing device, another consideration is that of numerical accuracy. On the 56001 DSP, the memory (and word) width is 24 bits. In a strictly analogue sense, this enables the processor to handle single word digital integer values with an accuracy of up to 144 dBs. Put into perspective, the best accuracy which can be held by a 16 bit value (or a 16 bit D-A or A-D converter) is 96 dBs. A dB, or decibel as it correctly known, is a way of measuring noise ratios. The higher this value is, then the least significant value (or error) becomes a smaller proportion of the overall result. If a 16 bit result is required on the output of a calculation, then up to 8 bits of noise can be accumulated before it starts to creep into the output value!

The Falcon DSP is provided with 96 kbytes of RAM on board, organised as 24 bit words. All of this can be used to store programs and sub-routines in the P memory area. To add a little confusion to the set up, the data memory, the X and Y areas, are split into two 16,000 word areas which actually overlap the lower and upper halves of the program space.

Falcon DSP memory map

From the programmer's point of view the model of the DSP looks something like this:



Falcon DSP - programmer's memory model

By looking carefully at the diagram above, it will be possible to see that there is one program area and two separate data areas, all of which start their virtual existence from an address of zero. What the diagram also tries to show is that despite the fact that the DSP has three memory areas, they do in fact share the same physical RAM. This is proved by the fact that writing a value into Y:0200 and reading from P:0200 would act on the same data, as would X:0200 and P:4200.

One final point to note is that the DSP features some on chip RAM and ROM. The lower 512 words of the program memory are in fact on board the chip itself and are not part of the 32k of external memory. This is all RAM. The first 40 words of which are reserved for use by the chip itself as interrupt and exception vectors of some type. The rest of the space is freely available for use by the DSP for program storage and execution.

The X: and Y: data areas share 512 words of RAM and a further 512 words of ROM split evenly between them. Each is configured as having its first 256 words as RAM and the next 256 words as ROM. These two areas of ROM contain pre-programmed maths tables, featuring full four quadrant sine wave and signal compression/decompression data.

What can the DSP do for me?

These chips find their way into all sorts of equipment nowadays, probably the most notable of these being the portable cellular telephone, the miniaturisation of which is due largely to the incorporation of DSPs where they are used to digitally encode/decode, analyse, filter and enhance the voice signal. Now, powerful as it is, it's pretty unlikely that you will find a Falcon built into a telephone but the principle is the same. With the Falcon's CODEC chip built in, speech analysis or compression/de-compression does become a possibility, as do real-time sound effects and equalisation. Since the switch matrix exists within the machine, then it is possible that the device could be used to perform JPEG or even MPEG picture compression or decompression. This may be achieved by the DSP receiving compressed data from the 68030 which it then processes and sends back as packets of decompressed data.

DSP access

There is no direct method of access to the DSP. For example, it cannot examine data on the Falcon screen directly, process and place it straight back, neither can the Falcon processor read from, or write directly into the DSP's memory. All data must be transferred into or out of the DSP via its SSI port (accessible only through the external DSP socket), or its host port which is mapped into the Falcon's memory space. The Falcon does not have direct program support for the control of data flowing between the DSP and the SSI, but extensive control for the transfer of programs into the DSP and the flow of data across the host port are provided, these are described in closer detail in the DSP sub-system section.

DSP program control

Unlike the Falcon's microprocessor, the DSP does not have any program or operating system ROM. In this respect it is a fairly daft device. After power-on, the DSP will do nothing initially, it will just sit around waiting for a user program to down load a piece of DSP program code, where upon the Falcon's operating system will reset the DSP and re-boot it into life. From this point onwards the two devices can get on with pretty much whatever they like. Any program or data communication between the two must be made via the DSP host port, the Falcon's operating system has a variety of routines built into it which allow this sort of thing to occur in a number of ways. It would make sense to read through the list of calls carefully to determine which should be used.

DSP software

Since the arrival of a multitasking kernel for the Falcon, it is quite possible that a number of Falcon applications may wish to use the resources of the DSP at once. The DSP is not that sort of device, its speed and efficiency are compromised by such activities. For this reason the DSP should only be accessed via the Falcon operating system; an application that wishes to use the DSP must first see if the DSP is already locked by another application, if so it must wait until such time that it has become unlocked. When the DSP becomes available for use, the application should lock the device for itself, preventing another application from stealing it.

When an application has made a successful bid for and secured the DSP, it may then download its own operating program or subroutine into the DSP program memory. For obvious reasons, a DSP program must not be larger than the total free memory as returned by the `Dsp_Available` command. This value is returned as two separate amounts, one each for the X: and Y: data areas. Programmers should be made aware that this is all program space but a program which is to use both areas must appreciate that there will be an unusable area of memory at the top of the Y: memory area, the implication here is that this effectively cuts the P: memory area in half! Within the reserved subroutine area, individual DSP subroutines are limited to an absolute maximum size of 1024 words, this must include all program, relocation, fixed data and initialised workspace! Additionally however, each subroutine may make use of the 256 words of reserved BSS space in both of the X: and Y: data spaces, this should be used only for uninitialised data storage.

DSP Ability codes

Atari are promoting the concept of a system where code which is to be executed within the DSP is not owned by any one host process (because of the possibility of multi-tasking applications), indeed there is no reason in principle why routines to do specific tasks can not be shared between applications. In an attempt to provide an efficient code sharing system it is necessary to develop an effective code identification system.

Close inspection of the DSP support routines will reveal that some of the calls provide, or use, an *ability* code. this is basically a number which allows the Falcon to monitor the programs and subroutines which are currently being held within the DSP system and to try and prevent duplication of application code. This should reduce the amount of code transfer which occurs, whilst increasing the data throughput potential and minimising the latency between an application requesting the use of the DSP and getting some meaningful response from it.

Since the DSP is a very fast and efficient data processing device, its total performance is degraded by having to upload executable code each time an application requires some form of processing. For this reason the program and the stacked subroutines can each own, or apply for, an ability code. Before any effort is made to download some code into the DSP, the current Falcon application should pass an ability code (after having first requested one, if necessary, using the `Dsp_RequestUniqueAbility` routine) to the DSP support call `Dsp_InqSubrAbility` (or `Dsp_GetProgAbility`). If a routine which matches this code already exists within the DSP, then it will not be necessary to download another, the one with the returned handle will do the job!

Programming considerations

Writing subroutines

Some important points should be born in mind when writing subroutines:

First of all, the DSP does not support the program relocation features of the standard main processor. It is necessary therefore to write all subroutines with a small relocater program at the start of it (if needed). When using an assembler to generate a subroutine, it is necessary to generate code which is either entirely position independent (practically impossible given its architecture and instruction set) or position dependent starting from memory location zero ($0000). The code should then relocate itself into an executable form using an address which is passed to it at execution time.

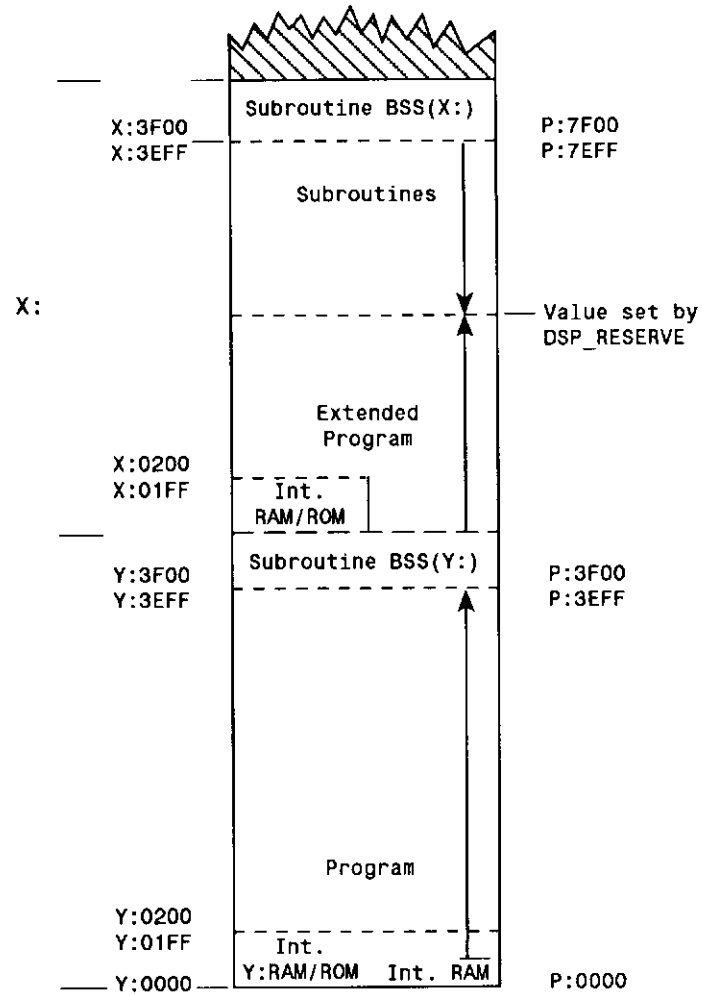
Since multiple subroutines can exist within the DSP, the Falcon's operating system may selectively remove routines to free up some space and then move the remaining ones around (usually closer to the top of the subroutine space). It is essential therefore, for the subroutine to check its current position against the execution address each time it is invoked, if the routine has been moved, then it will be necessary to relocate itself again.

The absolute maximum size of a subroutine is 1024 words, this includes all code (including the relocater) and all fixed data. Subroutines should not use more space than is absolutely necessary for its dynamic storage requirements.

Due to the nature of the mapping of the shared data and program spaces, any Y: area usage will cause a fragmentation of the P: space. It can be all to easy for the unwary subroutine programmer to overwrite space which is currently in use by a program, or even worse, to corrupt the program itself, thus causing it to crash the DSP at some later time. For this reason, two small reserved BSS areas exist within the memory map of the Falcons DSP. These total 256 words of data for each area (X: and Y:). This space is reserved for use by the subroutines and is intended for storing uninitialised data. It is important therefore, to note that after a subroutine has finished running, that another subroutine may choose to use the same BSS memory and overwrite previously held information. Since subroutines must check their relocation status when invoked, they must also re-establish their BSS data, otherwise it is likely to be meaningless!

Writing programs

The following considerations should be born in mind whilst writing programs which are to be launched into the area of RAM reserved for program use: due to the nature of the shared data and program spaces, careless use of X: or Y: data storage could prove to be disastrous to the execution of code within the DSP system. The following diagram shows how DSP memory is allocated on the Falcon:



Falcon DSP - Memory Map

Particular attention should be paid to the Y: area usage, since any attempt to write into this area will, in some way, cause a fragmentation of the P: space. Likewise, careless use of memory in the X: area could cause code or data within the subroutine area to become corrupted. As with writing subroutines, it can be all to easy for the unwary programmer to overwrite space which is currently in use by other code, with potentially disastrous consequences.

Programs should always be assembled into absolute format code. The DSP will always start a program by executing the instruction stored in the first location of the DSP vector table, location P:0000 (note that unlike the 680x0 processors, this is not an address; on the DSP these tables actually consist of a series of JMP instructions). Since the last element of this table is at location P:003F, then the first word of program space is location P:0040. Normally, when downloading code into the DSP, the Falcons main processor will automatically place the instruction JMP P:$0040 into location P:0000 for you, thus assuming that the program is to start execution from here. Such code would need to start off with a code fragment similar to the following:

```
        ORG     P:$0040
start   CLR A   #$7F,NO    * Code starts here...
        ...
```

It is possible however to force execution of code from a completely different address by writing your own start instruction into location P:0000. Such a program may start something like this:

```
start   equ     P:$0200
        org     p:0
        jmp     p:start    * Jump to code
        ...
        org     P:start
        CLR A   #$7F,NO    * Code starts here...
        ...
```

This code would load to, and start execution from address P:0200.

Chapter 2
The Operating System

GEMDOS and MiNT

MiNT implements the low level kernel for a multi-tasking operating system. The new calls for MiNT are made as additions to the GEMDOS part of the operating system. Programmers who are already familiar with UNIX and/or the POSIX.1 standard will find that MiNT provides the Atari range of machines with a very familiar (and portable) programming environment.

MultiTOS

The additions to the AES allow for a full multi window, multi tasking environment. It is now possible for many well written applications to work under MultiTOS with little or no modification. Atari have made other improvements to the GEM operating interface which will require at least some minor modifications to take advantage. These include a new 3D style user interface (including buttons, window slider bars and alert/dialog boxes) and a new popup menu system.

SpeedoGDOS

GDOS is the acronym used to describe the Graphics Device Operating System. This is the part of the operating system which, amongst other things, manages the use of fonts within the computer. This aspect of the system does much more than simply print the characters on the screen, it controls the memory usage for storing or caching the fonts, it decides which fonts are being used most frequently and most recently in order to keep the speed of font usage to an optimum and it is almost entirely responsible for all device independent output. This covers everything from printing fonts on the screen, to outputting the fonts and graphics to a dot matrix printer (at say 180 dots per inch), a laser printer (at 300 dots per inch) or maybe even to a Linotronic (a professional, very high resolution printing device) at 1270 or 2540 dots per inch. SpeedoGDOS is the name given to the latest incarnation of GDOS for Atari machines.

A brief history of Atari GDOS

The first font manager system that was introduced by Atari was a bitmap font manager simply called GDOS. This system was used by many development houses, but for one reason or another, some of them implemented fonts of different aspect ratios to those recommended by Atari. As a result, many applications could use GDOS, but some required a separate GDOS installation for the text to appear to be formatted correctly on screen, or to be printed correctly on an output device. This meant that the common resources of fonts (which as bitmaps could take large amounts of disk space), were never really common at all and were in fact unique to each application, worse still, the user had to reconfigure the ASSIGN.SYS file and re-boot the machine each time the graphics, DTP or word-processing application was changed. Another problem with GDOS was that since it used bitmap fonts exclusively, each font required different definitions for every point size required by the user, so not only were the individual fonts very large, but a full font set was potentially massive. This resulted in a very inefficient use of the hard disk space, compounded by the fact that the font set was ultimately limited to the number of point sizes which it could contain. The final blow was that not only was one font required for use on the screen, but another may be required for use by each different output device (where a dot matrix printer may be 180 to 360 DPI and a laser printer 300 DPI). Needless to say, the higher the resolution of the device, the larger the font set required to support it.

Technically speaking, GDOS is now dead and it has been replaced by a new system which not only supports the old bitmap fonts but also performs font scaling with a new type of font set. This new system is known as SpeedoGDOS. Speedo has been designed by a company called Bitstream which has a long history of using this sort of software technology and has established a massive range of Bitstream fonts, all of which are ready for use on the Atari range of computers.

What is Speedo?

At its lowest level, Speedo is a nice replacement to GDOS. Well written GDOS applications should be 100% compatible with Speedo. At its highest level, Speedo offers extra features which were never really available before such as the ability to select any point size from within a suitably written application, including sizes such as 10.25 point for example. Another advantage of this font scaling technique is that the same font definition can be used for each output device, gone is the requirement for a separate definition of each Swiss 10, 12, 14, 18, 24 and 36 point sizes for the screen and another set for an Epson LQ80 and another set for an HP Laserjet etc., one file suits all.

Speedo also provides the ability to draw true bézier curves so programmers of graphics programs can take advantage of this new facility which has the advantage of being 100% compatible with the Speedo device drivers, relieving the programmer of the otherwise tedious task of having to form the image in the computer manually before sending it to an output device of some type.

FontGDOS

As a final point, it is worth knowing that another version of GDOS exists, this is simply called FontGDOS. This GDOS is in fact Speedo, with all its Bézier drawing and bitmap facilities, but with all of the outline font manager removed. So strictly speaking then, FontGDOS comes in between the old GDOS and Speedo in terms of its functionality. The previous section described some of the superior features of an outline font system, what it did not point out however is that outline font managers can become less efficient and far less visually effective than standard bitmap fonts when dealing with small printer or screen character sets.

The reasons for this become evident when a scaleable font becomes so small that it is in effect reduced to little more than a blob on the screen. Bitmap fonts become very much more attractive at this point since each point size is individually tailored to show the required detail for the screen resolution in which they are currently running. It is also fair to say that they are faster to use too, since the individual characters do not require any form of calculation, they can simply be held in a modestly sized font cache within the machine and blasted onto the screen at the highest speed that the processor can manage.

So FontGDOS then is an update to the old GDOS system with the addition of bézier drawing facilities, but with the advantage of an outline font manager removed. FontGDOS may be more compatible with some older style applications than Speedo where the outline management may be of little or no use.

Chapter 3
The Video Sub-system

The video hardware in Falcon030 is supported by a complementary set of XBIOS system calls. These allow the user to control virtually any aspect of the display hardware configuration.

The system calls

Montype — Inquire attached monitor type

BASIC 2 FUNCTION Montype%

Devpac 3 montype.W
(stack 2 bytes)

Lattice C int Montype(void);

This system inquiry will return the type of display device which is currently connected to the computer. The following returns are possible:

0	ST monochrome monitor
1	ST colour monitor
2	VGA monitor
3	Television.

VgetRGB — Inquire palette entries

BASIC 2 SUB VgetRGB(BYVAL index%, BYVAL count%, BYVAL array&)

Devpac 3 array.L, count.W, index.W, vgetrgb.W (stack 10 bytes)

Lattice C void VgetRGB(int index, int count, long *array);

This call reads a range of values from the current palette, starting at `index`, with `count` values into `array`. `array` is then a block of long words which contain bytes of X-R-G-B data; applications which want to examine a single colour at a time are better off using the VDI call `vq_color`.

VgetSize — Inquire video mode memory size

BASIC 2 FUNCTION VgetSize&(BYVAL modecode%)

Devpac 3 modecode.W, vgetsize.W (stack 4 bytes)

Lattice C long VgetSize(int modecode);

Given a `modecode` (as discussed above), this call returns the amount of video RAM necessary to accommodate that screen. The returned size is in bytes and may be used directly in an appropriate `Mxalloc` call.

VsetMask — Get/Set VDI mask/overlay mode

BASIC 2 SUB VsetMask(BYVAL or_mask&, BYVAL and_mask&, BYVAL overlay%)

Devpac 3 overlay.L, and_mask.L, or_mask.W, vsetmask.W (stack 8 bytes)

Lattice C void VsetMask(long or_mask, long and_mask, int overlay);

This call is used exclusively in true colour screen modes; its purpose is to set the AND and OR masks used by the VDI in the calculation of `vs_color`. In practice, this enables or disables the use of the `X` or overlay bit in true colour graphic modes.

The default values for the masks are $FFFFFFFF (and_mask) and $00000000 (or_mask), forcing the X bit to zero in drawing operations. Values of $FFFFFFFF (and_mask) and $00000020 (or_mask) will set the X in drawing operations thus making the colour transparent to suitably connected external hardware. Finally, to clear the overlay , use masks $FFFFFFDF (and_mask) and $00000000 (or_mask).

If `overlay` is non-zero, then the system will operate in overlay graphics mode. If this value is zero, then the graphics system will be taken out of overlay mode.

Vsetmode — *Set video mode*

BASIC 2 FUNCTION Vsetmode%(BYVAL modecode%)

Devpac 3 modecode.W, vsetmode.W
(stack 4 bytes)

Lattice C int Vsetmode(int modecode);

This call is used to set a screen mode directly using a `modecode` as described in the video sub-system overview of this guide. This call is for use only by the experienced programmer since it does not automatically reallocate or request memory to cope with changes in the physical size of screen. Worse still, the VDI will not be informed of the screen modifications, in which case any attempt to use the VDI will appear as rubbish on the display.

One final point to note is that a call with an argument of -1 is treated by the OS as an inquiry as to the current screen's modecode. This can be called prior to a `VsetScreen` which will change parameters in some way. The returned value can then be used in a subsequent `VsetScreen` to restore the screen to its initial settings.

VsetRGB — Set palette entries

BASIC 2 SUB VsetRGB(BYVAL index%, BYVAL count%, BYVAL array&)

Devpac 3 array.l, count.W, index.W, vsetrgb.W (stack 10 bytes)

Lattice C void VsetRGB(int index, int count, long *array);

This call allows a range of values within the current palette to be changed, starting at `index`, with `count` values from the data `array`. The array is a block of long words which contain bytes of X-R-G-B data; applications which require to set a single colour at a time are better off using the VDI call `vs_color`.

VsetScreen — Set video mode

BASIC 2 FUNCTION VsetScreen%(BYVAL log&, BYVAL phys&, BYVAL res%, BYVAL modecode%)

Devpac 3 modecode.W, res.W, phys.L, log.L, setscreen.W (stack 14 bytes)

Lattice C int VsetScreen(void *log, void *phys, int res, int modecode);

This system call has been added to HiSoft languages as an updated version of the `Setscreen` call (note that assembly language programmers need no new 'binding'). It is now enhanced on the Falcon to be compatible with the old ST style call and yet be powerful enough to be useful with the new screen modes. The difference is that a fourth parameter has been added to the parameter list so that if `res` is set to 3 (an illegal screen mode on the ST and TT) then a fourth parameter is expected in the list. This value must be a word length bit mask or `modecode` which is described in the video sub-system overview.

Another feature of this call is that if a value of zero (`NULL`) is passed in both the `log` and `phys` parameter fields, then the operating system will automatically perform the relevant `Malloc` to request the appropriate amount of memory from the GEMDOS.

Programmers should note that careless use of this call can be dangerous. Changes made with this call will be recognised by the VDI but not the AES. Any attempt to use the AES in an application after having changed the screen in this way, will almost inevitably cause the program to crash. Only the VDI should be used for graphical output in such circumstances.

VsetSync — *Set external/internal sync mode*

BASIC 2 SUB VsetSync(BYVAL code%)

Devpac 3 code.W, vsetsync.W
(stack 4 bytes)

Lattice C void VsetSync(int code);

The computer contains a piece of circuitry which generates the required video timing signals which determine the total number of pixels on the screen. This is called the *dot clock*. By injecting a different signal into the appropriate socket on the rear of the machine, and in conjunction with the appropriate video programming, it is possible to customise the screen modes. This call switches the influence of the external pulses on or off as required. The bits used are as follows:

00000000 00000HVC

C	external Clock enable
V	external Vertical sync enable
H	external Horizontal sync enable

Chapter 4
The Audio Sub-system

The Falcon's audio subsystem has a series of comprehensive support subroutines which allow control of practically every aspect of sound recording/playback and CODEC, DSP and DMA device connection. Much of the power of the new Falcon hardware is hidden in these new calls.

The system calls

Buffoper Set sound play/record enable/disable

BASIC 2 FUNCTION Buffoper&(BYVAL mode%)

Devpac 3 mode.W, buffoper.W
(stack 4 bytes)

Lattice C long Buffoper(int mode);

This bitmap `mode` value is used to set or read the status of the record or play buffers. If `mode` is set to -1 then the value returned is the current bitmap status. In the bitmap, a value of 0 = OFF, 1 = ON. The bits are;

7	6	5	4	3	2	1	0
0	0	0	0	RR	RE	PR	PE

Where: RR=Record Repeat, RE=Record Enable, PR=Play Repeat and PE=Play Enable.

The sound system contains a 32 byte FIFO. When transferring data to the record buffer, software must look to see if the RE bit has been cleared by the hardware. If the bit was cleared then the FIFO has been flushed otherwise it must be flushed manually by clearing the RE bit.

Buffptr — Find sound system status structure

BASIC 2 FUNCTION Buffptr&(BYVAL pointer&)

Devpac 3 pointer.L, buffptr.W
(stack 6 bytes)

Lattice C long Buffptr(long *pointer);

This function returns the current position of the play and record data buffer pointers in the 4 longwords pointed to by `pointer`. These show the current status of the record and play buffer pointers; this can also be used to determine how much of a record buffer has been used (see `Buffoper`). The offsets from pointer are:

Offset	Function
$00	Play buffer pointer
$04	Record buffer pointer
$08	Reserved
$0C	Reserved

Devconnect — Configure switch matrix

BASIC 2 FUNCTION Devconnect&(BYVAL source%, BYVAL dest%, BYVAL srcclk%, BYVAL prescale%, BYVAL protocol%)

Devpac 3 protcol.W, prescale.W, srcclk.W, dest.W, source.W, devconnect.W
(stack 10 bytes)

Lattice C long Devconnect(int source, int dest, int srcclk, int prescale, int protocol);

This call is used to determine the connection of a source device to at least one destination device on the audio system switch matrix. Setting of the source clock prescaler and transmission protocol is also achieved. `source` determines the input device:

source	Input device
0	DMA playback
1	DSP transmit
2	External input
3	ADC (Microphone input or PSG)

The (potentially multiple) output devices are determined by the dest bitmap:

dest	Mask
$01	DMA record
$02	DSP receive
$04	External output
$08	DAC

srcclk determines the clock which used by the audio sub-system:

srcclk	Device clock speed
0	Internal 25.175MHz clock
1	Externally supplied clock
2	Internal 32.0MHz

prescale is used to determine the sample rate; the actual sample rate is calculated as 'the selected clock speed divided by 256, divided by the prescaler'. The prescale value is N-1, where N can be between 1 and 12. A value of N>12 will mute the CODEC in which case the Sndstatus command can be used to reset the CODEC. A value of 0 will cause the system to use the /1280, /640, /320, /160 prescale values mentioned in the Soundcmd documentation.

The following table is a list of all of the device clock frequencies which can be generated using the 25.175MHz system clock, given a particular prescale value:

Prescale	Division ratios 25.175MHz/(98340)	Resulting Clock (Hz)
1	+512 / (+2)	49170
2	+768 / (+3)	32780
3	+1024 / (+4)	24585
4	+1280 / (+5)	19668
5	+1536 / (+6)	16390
6	+1792 / (+7)	14049*
7	+2048 / (+8)	12292
8	+2304 / (+9)	10927*
9	+2560 / (+10)	9834
10	+2816 / (+11)	8940*
11	+3072 / (+12)	8195
12	+3328 / (+13)	7565*

Note that the frequencies marked * are not valid for use with the CODEC. A similar table of frequencies would have to drawn up using the given division ratios if an external clock source were connected to the DSP port and selected using the `srcclk` = 1 parameter.

`protocol` enables or disables the handshaking mode; it should be 0 to enable handshaking or 1 to disable it.

Dsptristate — *Set DSP tristate mode*

BASIC 2 FUNCTION Dsptristate&(BYVAL dspxmit%, BYVAL dsprec%)

Devpac 3 dsprec.W, dspxmit.W, dsptristate.W (stack 2 bytes)

Lattice C long Dsptristate(int dspxmit, int dsprec);

This function removes the DSP's SSI transmit and receive ports from the sound systems data matrix. The arguments passed to the call are status flags in both cases, where a value of 0 means tristate (isolated) mode and 1 means enable (on the bus) mode.

Gpio — *Program GP output pins*

BASIC 2 FUNCTION Gpio&(BYVAL mode%, BYVAL data%)

Devpac 3 data.W, mode.W, gpio.W (stack 2 bytes)

Lattice C long Gpio(int mode, int data);

The Falcon's external DSP port contains three user programmable general purpose (GP) input/output pins. This system call allows the programmer access to these pins in order to be able to set their direction as either input or output and to actually read or write the pins status. `mode` values are:

0	Set the I/O direction; `data` gives the I/O direction for each of the three pins: 1 for output, 0 for input
1	Read the pins status
2	Set the pins status to data

The function returns the current status of the pins for `mode` = 1 or zero otherwise.

Locksnd — Lock sound system

BASIC 2 FUNCTION Locksnd&()

Devpac 3 locksnd.W
(stack 2 bytes)

Lattice C long Locksnd(void);

Since MultiTOS will allow multiple sound utilities to run concurrently, an application which requires use of the sound system must not simply assume that the hardware is available for use at any given time. This system call enables an application to attempt to claim the hardware for its own use and will be given a return of 1 if successful. If the call failed to claim the hardware because it is already in use, then an error code of -129 (SNDLOCKED) will be returned.

Setbuffer — Set sound play/record buffers

BASIC 2 FUNCTION Setbuffer&(BYVAL rec%, BYVAL begaddr&, BYVAL endaddr&)

Devpac 3 endaddr.L, begaddr.L, rec.W, setbuffer.W
(stack 12 bytes)

Lattice C long Setbuffer(int rec, void *begaddr, void *endaddr);

This function is used to set-up the sample play/record buffers. `rec` determines if the buffer is to be set to record or play, 0 sets playback, 1 sets record. `begaddr` is the start address of the buffer, `endaddr` is the first location after the end of the buffer.

Setinterrupt — Set sound end interrupt

BASIC 2 FUNCTION Setinterrupt&(BYVAL source%, BYVAL cause%)

Devpac 3 cause.W, source.W, setinterrupt.W (stack 6 bytes)

Lattice C long Setinterrupt(int source, int cause);

It is possible for the system to generate an interrupt at the end of a sample record or playback frame. Used in conjunction with the frame repeat bit, seamless joins between double buffered sounds become possible. The interrupt source can be programmed to be either the MFP timer A (`source` = 0) or MFP interrupt 7 (`source` = 1). The cause of the interrupt can be: no interrupt (`cause` = 0), the end of a play buffer (1), the end of a record buffer (2) or the end of a play or record buffer (3).

Setmode — Set sample playback resolution

BASIC 2 FUNCTION Setmode&(BYVAL mode%)

Devpac 3 mode.W, setmode.W (stack 2 bytes)

Lattice C long Setmode(int mode);

The sample playback resolution can be determined using this call; note that 16 bit mono is *not* a legal operation. Legal `mode` values are:

Value	Mode
0	8 bit stereo
1	16 bit stereo
2	8 bit mono.

Note that this call sets *playback* mode; recording can only be achieved in 16 bit stereo.

Setmontrack — *Set internal track*

BASIC 2 FUNCTION Setmontrack&(BYVAL montrack%)

Devpac 3 montrack.W, setmontrack.W
(stack 4 bytes)

Lattice C long Setmontrack(int montrack);

The Falcon's internal speaker can be made to play any one of the four possible sound tracks. The value `montrack` can be any value between 0 and 3.

Settrack — *Set play/record tracks*

BASIC 2 FUNCTION Settrack&(BYVAL playtracks%, BYVAL rectracks%)

Devpac 3 rectracks.W, playtracks%, settrack.W
(stack 6 bytes)

Lattice C long Settrack(int rectracks, int playtracks);

This function is used to set the number of record and playback tracks. Please note that a track is considered to be stereo, for this reason when the hardware is working in 8 bit stereo mode, two samples are read in at a time.

Sndstatus — *Inquire sound system status*

BASIC 2 FUNCTION Sndstatus&(BYVAL reset%)

Devpac 3 reset.W, sndstatus.W
(stack 4 bytes)

Lattice C long Sndstatus(int reset);

If `reset` = 0 then this call returns a bitmap which presents the current status of the CODEC, which is returned in the lower nybble. The L & R bits show if an overflow of some form has occurred in either the analogue to digital or filtering processes. The bits are as follows;

7	6	5	4	3	2	1	0
0	0	L	R	S	S	S	S

These bits are:

0	No error
1	Invalid control field (data still assumed to be valid)
2	Invalid sync format. This will cause a MUTE condition.
3	Serial clock is out of range. A MUTE condition will occur.
R	If set - Right channel clipping has occurred.
L	If set - Left channel clipping has occurred.

If `reset` is 1, then after making this call the sound sub-system is forced into the following state:

The DSP is tristated, gain and attenuation are set to zero, matrix connections are reset, adderin is disabled, mode is set to 8 bit stereo, play/record tracks are set to track 0, monitor track is set to zero, interrupts are disabled, buffer operation is disabled (0).

Soundcmd — Issue command to sound system

BASIC 2 FUNCTION Soundcmd&(BYVAL mode%, BYVAL data%)

Devpac 3 data.W, mode.W, soundcmd.W
(stack 6 bytes)

Lattice C long Soundcmd(int mode, int data);

This call can be used to set, or enquire about the current settings of, a range of different aspects of the sound hardware. For example, `mode` tells the call which parameter is to be changed and `data` is what it is to be changed to. If `mode` is a negative number then the current settings are returned instead. The legal mode and data values are as follows:

Mode	Operation	Description
0	LTATTEN	Sets the left channel output attenuation which is measured in -1.5 dB increments.
		Bit format: xxxx xxxx LLLL xxxx
1	RTATTEN	Sets the right channel output attenuation which is measured in -1.5 dB increments
		BIT format: xxxx xxxx RRRR xxxx
2	LTGAIN	Sets the left channel input gain in 1.5 dB steps
		Bit format: xxxx xxxx LLLL xxxx

3	RTGAIN	Sets the right channel input gain in 1.5 dB steps Bit format: xxxx xxxx RRRR xxxx
4	ADDERIN	Set the 16 bit signed adder to receive its input from the ADC, matrix or both. The input to this function is a bitmap: 76543210 xxxxxxMA where: M = matrix A = ADC
5	ADCINPUT	The input to the ADC can be set to be the left and right channels of the PSG or the microphone inputs. A 0 in a bit sets microphone input, a 1 will set the input to be from the PSG. 76543210 xxxxxxLR where: L = left input R = right input
6	SETPRESCALE	The value set by this call will act as the clock pre-scale value when the Devconnect pre-scale value is set to 0. Values are" 0 Invalid 1 divide by 640 2 divide by 320 3 divided by 160

Unlocksnd — *Unlock sound system*

BASIC 2 FUNCTION Unlocksnd&()

Devpac 3 unlocksnd.W
(stack 2 bytes)

Lattice C long Unlocksnd(void);

This call must be made to release the hardware back to the system at some stage. An error code of -128 (SNDNOTLOCK) will be returned if the hardware was not already locked out, otherwise it will return 0.

Chapter 5
The DSP Sub-system

The DSP sub-system features a number of special operating system support calls. These fall into two areas, those used for the control of data which is passed between the DSP and the host and those used for the control of programs and subroutines which are run on the DSP.

The system calls

Data transfer control

Dsp_BlkBytes — Send bytes from/to DSP

BASIC 2 SUB Dsp_BlkBytes(BYVAL data_in&, BYVAL size_in&, BYVAL data_out&, BYVAL size_out&)

Devpac 3 size_out.L, data_out.L, size_in.L, data_in.L, dsp_blkbytes.W
(stack 18 bytes)

Lattice C void Dsp_BlkBytes(unsigned const char *data_in, long size_in, unsigned char *data_out, long size_out);

This call handshakes a block of bytes to the DSP; the data is *not* sign extended before transmission. On receipt of data from the DSP, only the low byte of the transfer register is read. `data_in` and `data_out` are 8 bit byte arrays. `size_in` and `size_out` are limited to a maximum length of 64 kbytes each.

Dsp_BlkHandShake Handshake data from/to DSP

BASIC 2 SUB Dsp_BlkHandShake(BYVAL data_in&,
BYVAL size_in&, BYVAL data_out&, BYVAL size_out&)

Devpac 3 size_out.L, data_out.L, size_in.L, data_in.L,
dsp_blkhandshake.W
(stack 18 bytes)

Lattice C void Dsp_BlkHandShake(const void *data_in, long size_in,
void *data_out, long size_out);

This call handshakes a block of DSP word to/from the DSP. To initiate a transfer, simply point `data_in` and `data_out` to the start addresses of their respective transmit or receive buffers, provide the length of the send and receive buffers and then make the call; by making one of the length values zero, it is possible to use this call to transfer data in one direction only.

Despite the fact that `size_in` and `size_out` are long word values, you should be aware of the fact that a 64k limit is imposed on the size of transferred data blocks.

Dsp_BlkUnpacked Send longs from/to DSP

BASIC 2 SUB Dsp_BlkUnpacked(BYVAL data_in&,
BYVAL size_in&, BYVAL data_out&, BYVAL size_out&)

Devpac 3 size_out.L, data_out.L, size_in.L, data_in.L,
dsp_blkunpacked.W
(stack 18 bytes)

Lattice C void Dsp_BlkUnpacked(const long *data_in, long size_in,
long *data_out, long size_out);

Once again, this routine is a block transfer call. The input format of the data is always assumed to be a 32 bit word, however, depending on the result from the call `Dsp_GetWordSize`, only the correct number of the least significant bytes are transmitted from storage. For this reason, this call can only be made on machines which return a word size value of 4 or less! For example, if the word size is declared to be 3 bytes (the 56001 DSP chip is a 24 bit device) then only the least significant 3 bytes of data are transmitted from each of the 4 byte data packets. The same principles apply to data received back from the DSP. An array of 4 byte values will be filled with data from the DSP with all the bits right justified (or forced into the least significant bytes).

It must be noted, however, that any redundant bits at the most significant end of the long word will not necessarily be cleared and will probably contain meaningless data. For this reason it is very much the responsibility of the calling program to decide how to treat the returned packets i.e. to mask off the redundancy in the most significant bits if required. `size_in` and `size_out` are limited to a maximum length of 64 kbytes each.

Dsp_BlkWords — Send words from/to DSP

BASIC 2 SUB Dsp_BlkWords(BYVAL data_in&, BYVAL size_in&, BYVAL data_out&, BYVAL size_out&)

Devpac 3 size_out.L, data_out.L, size_in.L, data_in.L, dsp_blkwords.W
(stack 18 bytes)

Lattice C void Dsp_BlkWords(const short *data_in, long size_in, short *data_out, long size_out);

This command is used to send a block of 16 bit words to the DSP. As each word is taken from the buffer, it is sign extended from 16 bits to the word length of the receiving device (usually 24 bit) before being passed onto the DSP itself. Data which is received back from the DSP is truncated to the two least significant bytes before being placed into the receive buffer. `size_in` and `size_out` are limited to a maximum length of 64 kbytes each.

Dsp_DoBlock — Stream data from/to DSP

BASIC 2 SUB Dsp_DoBlock(BYVAL data_in&, BYVAL size_in&, BYVAL data_out&, BYVAL size_out&)

Devpac 3 size_out.L, data_out.L, size_in.L, data_in.L, dsp_doblock.W
(stack 18 bytes)

Lattice C void Dsp_DoBlock(const void *data_in, long size_in, void *data_out, long size_out);

This routine offers a relatively fast and flexible way to transfer data backwards and forwards between the host and the DSP. Please note that the `size_in` and `size_out` values are expressed in *DSP* words and not necessarily in the word size of the host. The number of bytes which constitute a DSP word can be obtained using the `Dsp_GetWordSize` call.

To initiate a transfer, simply point `data_in` and `data_out` to the start addresses of their respective transmit or receive buffers, provide the length of the send and receive buffers and then make the call. The host will then sit and wait for the DSP to collect the first item of data from the `data_in` buffer, where upon it will then proceed to feed data to the DSP, assuming that the DSP is collecting it! There is no handshaking using this call so it is possible for data to go missing if the DSP is too heavily burdened to cope with the input data rate. When finished, the routine will then wait for the DSP to start sending data back. The routine will read out the requested number of data words into the buffer even if the DSP does not send the full amount! Finally, by making one of the length values zero, it is possible to use this call to transfer data in one direction only.

Despite the fact that `size_in` and `size_out` are long word values, you should be aware of the fact that a 64k limit is imposed on the size of transferred data blocks.

Dsp_GetWordSize — *Obtain DSP word size*

BASIC 2 FUNCTION Dsp_GetWordSize%()

Devpac 3 dsp_getwordsize.W
(stack 2 bytes)

Lattice C int Dsp_GetWordSize(void);

This call returns a value which represents the size of a word of data for the DSP currently connected to the system. A value of three represents 3 bytes or more precisely 24 bits of data per DSP word. This value must be used by applications which use the DSP, to ascertain the size of data blocks which are transferred back and forth between the two devices. Buffer sizes must be modulo this value. The value which is returned could change at a later date, especially if the type or nature of the DSP were to change in future Falcon'esque' machines.

Dsp_InStream Submit data to DSP input dæmon

BASIC 2 SUB Dsp_InStream(BYVAL data_in&, BYVAL block_size&, BYVAL num_blocks&, VARPTR blocks_done&)

Devpac 3 blocks_done.L, num_blocks.L, block_size.L, data_in.L, dsp_instream.W
(stack 18 bytes)

Lattice C void Dsp_InStream(const void *data_in, long block_size, long num_blocks, long *blocks_done);

This routine is primarily designed to allow the transmission of multiple blocks of data without the repeated use of a `Dsp_DoBlock` call. Operation of the transfer is largely transparent to the calling routine, allowing it to get on with other work in the mean time. The transfer of the next block in the queue is triggered by an interrupt from the DSP, but the actual block transfer itself is executed without handshaking. When all the blocks have been sent, the routine will tell the DSP not to send any more data request interrupts. The calling routine can check the progress of the transmission of the data blocks by examining the variable `blocks_done` at any time. The job is finished when `blocks_done` becomes equal to `num_blocks`.

Dsp_IOStream Transfer DSP data via I/O dæmons

BASIC 2 SUB Dsp_IOStream(BYVAL data_in&, BYVAL data_out&, BYVAL block_insize%, BYVAL block_outsize&, BYVAL num_blocks&, VARPTR blocks_done&)

Devpac 3 blocks_done.L, num_blocks.L, block_outsize.L, block_insize.L, data_out.L, data_in.L, dsp_iostream.W
(stack 26 bytes)

Lattice C void Dsp_IOStream(const void *data_in, void *data_out, long block_insize, long block_outsize, long num_blocks, long *blocks_done);

`Dsp_IOStream` is similar to a combination of `Dsp_InStream` and `Dsp_OutStream`, but it does make one very important assumption and that is that as soon as the DSP is ready to transfer one block of data, it will also be ready to accept another as input.

To initiate a transfer, simply point `data_in` and `data_out` to the start addresses of their respective transmit or receive buffers, `block_insize` gives the size of blocks sent to the DSP, `block_outsize` gives the size of block which will be received from the DSP. `num_blocks` is the total number of blocks to be transferred, `blocks_done` tracks the blocks as they are transferred.

Dsp_MultBlocks
Transfer struct dspblocks from/to DSP

BASIC 2 SUB Dsp_MultBlocks(BYVAL numsend&, BYVAL numreceive&, BYVAL sendblocks&, BYVAL receiveblocks&)

Devpac 3 receiveblocks.L, sendblocks.L, numreceive.L, numsend.L, dsp_multblocks.W
(stack 18 bytes)

Lattice C void Dsp_MultBlocks(long numsend, long numreceive, const void *sendblocks, void *receiveblocks);

This call provides the programmer with the ability to send and/or receive multiple packets of data between the DSP and the host with a single operating system call. The number of blocks to be sent or received are declared in `numsend` and `numreceive`. The blocks of data to be sent are listed in two arrays or lists called `sendblocks` and `receiveblocks`. For C programmers these are structures of the type:

```
struct dspblock {
    short blocktype;     0 = 32 bit longs
                         1 = signed 16 bit words
                         2 = unsigned 8 bit bytes
    long blocksize;
    void *blockaddr;
};
```

In each case, the array is a list of the blocks to be sent or received. The first element of each array is the type of data which the block represents, i.e. longs, signed words or bytes. Second, the number of elements in the block and finally, the address where the block to be sent or into which the data is to be placed, can be found in the computer's memory.

Dsp_OutStream Get data from DSP output dæmon

BASIC 2 SUB Dsp_OutStream(BYVAL data_out&, BYVAL block_size&, BYVAL num_blocks&, VARPTR blocks_done&)

Devpac 3 blocks_done.L, num_blocks.L, block_size.L, data_out.L, dsp_outstream.W
(stack 18 bytes)

Lattice C void Dsp_OutStream(void *data_out, long block_size, long num_blocks, long *blocks_done);

This routine is primarily designed to allow the reception of multiple blocks of data without the repeated use of a `Dsp_DoBlock` call. Operation of the transfer is largely transparent to the calling routine, allowing it to get on with other work in the mean time. The transfer of the next block in the queue is triggered by an interrupt from the DSP, but the actual block transfer itself is executed without handshaking. When all the blocks have been received, the routine will tell the DSP not to send any more data request interrupts. The calling routine can check the progress of the transmission of the data blocks by examining the variable `blocks_done` at any time. The job is finished when `blocks_done` becomes equal to `num_blocks`.

Dsp_RemoveInterrupts Remove DSP vectors

BASIC 2 SUB Dsp_RemoveInterrupts(BYVAL mask%)

Devpac 3 mask.W, dsp_removeinterrupts.W
(stack 4 bytes)

Lattice C void Dsp_RemoveInterrupts(int mask);

`Dsp_RemoveInterrupts` stops the DSP from generating ready to receive or ready to send interrupts to the host. `mask` is an 8 bit mask which represents the interrupt to turn off, where the bits are:

1	Do not generate interrupts when there is data ready for the host
2	Do not generate interrupts when the DSP requires data from the host
3	Disable generation of interrupts of both types.

The primary use of this call is to terminate one or other of the receive or transmit interrupts which were originally set up by a `Dsp_SetVectors` where a number of data elements were expected by the host but fewer transactions actually occurred.

Dsp_SetVectors — *Set DSP interrupt vectors*

BASIC 2 SUB Dsp_SetVectors(BYVAL receiver&, BYVAL transmitter&)

Devpac 3 transmitter.L, receiver.L, dsp_setvectors.W (stack 10 bytes)

Lattice C void Dsp_SetVectors(void (*__stdargs receiver)(long), long (*transmitter)(void));

`Dsp_SetVectors` allows the host process to install a pair of functions which are called when an interrupt is received from the DSP. `receiver` should point to a function that the user wishes to be called when the DSP has data to be sent to the host process. The datum received from the DSP is passed to the receiver routine is as a longword on the stack, the lower three bytes of which are valid.

`transmitter` should point to a routine which is to be called when the DSP requests data. If the transmitter function returns a non-zero long value, the XBIOS portion of the interrupt handler will send the low three bytes of the longword to the DSP. No data will be sent if the 32 bit long word which is returned is a 0; to send back a zero DSP word, OR in a value into the high byte of the returned value.

If either `receiver` or `transmitter` are 0 (NULL), the corresponding interrupt will not be enabled. The host must remove its interrupts by using the `Dsp_RemoveInterrupts` call.

Dsp_Available — Inquire available DSP memory

BASIC 2 SUB Dsp_Available(VARPTR xavailable&, VARPTR yavailable&)

Devpac 3 yavailable.L, xavailable.L, dsp_available.W (stack 10 bytes)

Lattice C void Dsp_Available(long *xavailable, long *yavailable);

The amount of free ram available from within the DSP system is returned in the `xavailable` and `yavailable` variables (X: memory and Y: memory respectively). The free memory will start at physical address 0 in both address spaces but note that program space overlaps both the X: and Y: areas and that the low 64 words of space in the Y: area are used as interrupt vectors. This call would normally be used to determine the amount of free memory which is available for use by either a DSP program or a DSP resident subroutine in the following manner:

DSP subroutines are always loaded into and executed within the area of RAM which is known as X: memory (X:0 = P:16k). Each time a subroutine is launched within the DSP, the subroutine manager will place the new routine underneath the previous one. Therefore the value returned in `xavailable` is the size of the unused area of X: memory or the memory address immediately beneath the lowest stacked subroutine.

The major area of program space is that which is overlapped by the Y: memory space (Y:0 = P:0). Since only one program can be resident within the DSP at a time, current versions of the operating system will always return the same value in `yavailable`. This is always the length of the Y: memory less the decimal value of 256 which is the length of the subroutine BSS area located at the top of Y: memory, $3EFF. This BSS space is significant because programs actually run in 'program' memory which is both X: and Y: memory together, so in principle there is little to stop a real DSP hack from making a program use all of the Y: memory and extend it up into the bottom of the X: memory space. Care would be required here however since any subroutine activity may cause corruption of the program or its data by writing into its legitimate BSS memory.

Dsp_ExecBoot — Execute DSP boot program

BASIC 2 SUB Dsp_ExecBoot(BYVAL codeptr&, BYVAL codesize&, BYVAL ability%)

Devpac 3 ability.W, codesize.L, codeptr.L, dsp_execboot.W (stack 12 bytes)

Lattice C Dsp_ExecBoot(const void *codeptr, long codesize, int ability);

This routine will reset the DSP and download a boot program into the 512 words of internal DSP RAM. If the code is larger than the internal memory then the rest of the program will be ignored. The types of program which are downloaded using this call should be restricted to those whose intention is to entirely take over the DSP or to act as a debugger of some form. Applications which share the DSP as a resource with other programs must use the `Dsp_LoadProg` or `Dsp_ExecProg` calls instead. `codeptr` must point to the address of the block to be transferred within the host memory while `codesize` is the length of the file in DSP words.

Dsp_ExecProg — Execute loaded DSP program

BASIC 2 SUB Dsp_ExecProg(BYVAL codeptr&, BYVAL codesize&, BYVAL ability%)

Devpac 3 ability.W, codesize.L, codeptr.L, dsp_execprog.W (stack 12 bytes)

Lattice C void Dsp_ExecProg(const void *codeptr, long codesize, int ability);

This call will launch a DSP program into the DSP from a binary code block contained within the host memory which is pointed to by `codeptr`. The length of the program transferred is set by the variable `codesize` and should not exceed that set by a previous `Dsp_Reserve` call. The value of `codesize` is measured in DSP words.

Dsp_FlushSubroutines — Flush DSP subroutines

BASIC 2 SUB Dsp_FlushSubroutines()

Devpac 3 dsp_flushsubroutines.W
(stack 2 bytes)

Lattice C void Dsp_FlushSubroutines(void);

In cases where X: memory appears to be too small to run another subroutine, this call can be used to clear out all of the subroutines and return the memory to the pool. This call should only be used as a last resort since frequent flushing of often used subroutines will degrade the performance of the DSP.

Dsp_GetProgAbility — Get current program ability

BASIC 2 FUNCTION Dsp_GetProgAbility%()

Devpac 3 dsp_getprogability.W
(stack 2 bytes)

Lattice C int Dsp_GetProgAbility(void);

This call will provide the host with the ability code of the program which is currently residing and/or running in the DSP. If the returned code is not recognised by the caller, then the program is unlikely to do anything useful for it. In this instance the host has to assess whether or not it needs to remove the current program and replace it with another which will do the job.

Dsp_Hf0 — Read/Write HSR bit 3

BASIC 2 FUNCTION Dsp_Hf0%(BYVAL flag%)

Devpac 3 flag.W, dsp_hf0.W
(stack 4 bytes)

Lattice C int Dsp_Hf0(int flag);

This call will either read from or write to bit #3 of the Host Status Register (HSR). If `flag` is 0 or 1 then the relevant status will be written into the bit. If `flag` = $FFFF, then the current status of the flag will be returned without affecting its value.

Dsp_Hf1 — Read/Write HSR bit 4

BASIC 2 FUNCTION Dsp_Hf1%(BYVAL flag%)

Devpac 3 flag.W, dsp_hf1.W
(stack 4 bytes)

Lattice C int Dsp_Hf1(int flag);

Functionally identical to `Dsp_Hf0` only this call will operate on bit #4 of the HSR.

Dsp_Hf2 — Read HCR bit 3

BASIC 2 FUNCTION Dsp_Hf2%()

Devpac 3 dsp_hf2.W
(stack 2 bytes)

Lattice C int Dsp_Hf2(void);

This call returns the status of bit #3 of the Host Control Register (HCR). Note that this is a read only register and that the bit in question can only be set by the DSP.

Dsp_Hf3 — Read HCR bit 4

BASIC 2 FUNCTION Dsp_Hf3%()

Devpac 3 dsp_hf3.W
(stack 2 bytes)

Lattice C int Dsp_Hf3(void);

Similar to `Dsp_Hf2` except that bit #4 of the HCR is returned.

Dsp_HStat — *Read interrupt status register*

BASIC 2 FUNCTION Dsp_HStat%()

Devpac 3 dsp_hstat.W
(stack 2 bytes)

Lattice C int Dsp_HStat(void);

This call will return the status of the read only, byte wide DSP register, called the Interrupt Status Register (ISR). By reading the status of various bits contained within this register, it is possible to tell if the DSP host port is ready to transmit or receive data.

Bit	Name	Function
0	RXDF	Receive Data Register Full
1	TXDE	Transmit Data Register Empty
2	TRDY	Transmitter Ready
3	HF2	Host Flag 2
4	HF3	Host Flag 3
5	N/A	Reserved for later expansion
6	DMA	DMA Status
7	HREQ	Host Request

Dsp_InqSubrAbility — *Locate resident DSP subroutine*

BASIC 2 FUNCTION Dsp_InqSubrAbility%(BYVAL ability%)

Devpac 3 ability.W, dsp_inqsubrability.W
(stack 4 bytes)

Lattice C int Dsp_InqSubrAbility(int ability);

In an effort to see if a subroutine already exists within the DSP which has a specific ability, the host only needs to make this call. An attempt is made to match the ability code requested by the host with those already inside the DSP. If the handle which is returned is 0 then no routine currently exists, in which case the host will have to download one itself. However, if a routine does exist within the machine, then it may be called using the returned handle.

Dsp_LoadProg — Load & Execute DSP program

BASIC 2 FUNCTION Dsp_LoadProg%(BYVAL file$, BYVAL ability%, BYVAL buffer&)

Devpac 3 buffer.L, ability.W, file.L, dsp_loadprog.W (stack 12 bytes)

Lattice C int Dsp_LoadProg(const char *file, int ability, void *buffer);

This call will load a DSP routine from disk and execute the program automatically. The program must be stored as an ASCII .LOD format file and the length of the executable code must not be larger than that reserved for it using the `Dsp_Reserve` call. File should point to the file name (and where necessary, the path) of the file to be executed. `ability` is the 16 bit code which describes the functionality of the call (refer to the chapter on the DSP sub-system for further details). `buffer` must point to an address within the host where it can process the .LOD file and the DSP code which it generates. The size of this buffer is calculated as "3*(Length of program/data words + (3 * number of blocks in the program))". The return from this function indicates what happened during the call. A value of 0 means that the application was launched without problem. A return of -1 indicates that an error occurred sometime before the DSP file could be executed.

Dsp_LoadSubroutine — Load DSP subroutine

BASIC 2 FUNCTION Dsp_LoadSubroutine%(BYVAL ptr&, BYVAL size&, BYVAL ability%)

Devpac 3 ability.W, size.L, ptr.L, dsp_loadsubroutine.W (stack 12 bytes)

Lattice C int Dsp_LoadSubroutine(const void *ptr, long size, int ability);

This call will install a subroutine into the DSP memory for use at a later stage. `ptr` must point to the first location of a block of DSP subroutine code which is to be downloaded. The size of the subroutine in DSP words is sent in the variable `size` and the ability code of the routine is also sent as the third parameter. The handle returned by the routine will be a positive value if the subroutine was installed without any problems. This handle must be used in any future communication between the subroutine and the host. A handle return of 0 indicates that the subroutine was not installed. Subroutines which are installed will remain in the DSP until either they are overwritten by the subroutine manager or flushed out using the `Dsp_FlushSubroutines` call.

Dsp_Lock — Obtain exclusive lock on DSP

BASIC 2 FUNCTION Dsp_Lock%()

Devpac 3 dsp_lock.W
(stack 2 bytes)

Lattice C int Dsp_Lock(void);

This call should be made by a host application before attempting to use the DSP. This is because Falcon supports a multi-tasking operating system. Since a number of applications may now potentially be vying for the DSP at any given time, `Dsp_Lock` provides a useful mechanism for preventing those applications from accidentally hijacking the DSP from each other. The call returns a value of 0 if the application has made a successful bid to hire the use of the DSP and will remain the owner of the DSP until a `Dsp_Unlock` call is made at some later point. A return value of -1 indicates that the DSP is currently in use by some other process and that for the time being at least, the caller must either sit and wait its turn (presumably calling `Dsp_Lock` periodically) or if possible, it should go away do the processing itself.

Dsp_LodToBinary — Load .LOD file as binary

BASIC 2 FUNCTION Dsp_LodToBinary&(BYVAL file$,
BYVAL codeptr&)

Devpac 3 codeptr.L, file.L, dsp_lodtobinary.W
(stack 10 bytes)

Lattice C long Dsp_LodToBinary(const char *file, void *codeptr);

This routine will load a DSP .LOD program into memory and convert it into the binary format required for transmission to the DSP using the `Dsp_ExecProg` routine. The function will return the size of the program in DSP words. A negative value will indicate that an error occurred during execution of the call. Once again, `codeptr` should point to the first location of the block of memory into which the binary data is to be placed.

Dsp_RequestUniqueAbility — *Request ability code*

BASIC 2 FUNCTION Dsp_RequestUniqueAbility%()

Devpac 3 dsp_requestuniqueability.W
(stack 2 bytes)

Lattice C int Dsp_RequestUniqueAbility(void);

This call provides a mechanism by which a host process may request an identifier for a piece of code which is about to be downloaded. Any future inquiry about this routine must be made by referring to it by its ability code.

Dsp_Reserve — *Reserve DSP memory*

BASIC 2 FUNCTION Dsp_Reserve%(BYVAL xreserve&,
BYVAL yreserve&)

Devpac 3 yreserve.L, xreserve.L, dsp_reserve.W
(stack 10 bytes)

Lattice C int Dsp_Reserve(long xreserve, long yreserve);

Before downloading and running a program within the DSP system, it is first necessary to reserve sufficient space to contain the code. The size of the request must not be larger than that returned by the preceding `Dsp_Available` call. Please note that despite having made a reservation, the amount returned by `Dsp_Available` will not have changed. This is because the value returned is that which is not being used by subroutines. Programs are all deemed to use the same space where as a number of sub-routines can remain resident along side one another and must therefore be protected and allocated somehow. A return of 0 means that the request for memory has been granted, a -1 returned means that the request failed for some reason.

Dsp_RunSubroutine Run resident DSP subroutine

BASIC 2 FUNCTION Dsp_RunSubroutine%(BYVAL handle%)

Devpac 3 handle.W, dsp_runsubroutine.W
(stack 4 bytes)

Lattice C int Dsp_RunSubroutine(int handle);

This call will execute a subroutine resident within the DSP. Before the call is made however, it must have first been identified using either the `Dsp_InqSubrAbility` or `Dsp_LoadSubroutine` calls to obtain a handle. A negative status return indicates that the routine was not successfully launched, a status of 0 indicates that everything is fine.

Dsp_TriggerHC Trigger host command interrupt

BASIC 2 SUB Dsp_TriggerHC(BYVAL vector%)

Devpac 3 vector.W, dsp_triggerhc.W
(stack 4 bytes)

Lattice C void Dsp_TriggerHC(int vector);

The DSP features a number of special interrupts which are dedicated to communicating between the host processor and the DSP itself. Most of these interrupts are reserved by the computer for its own use (many of the XBIOS routines already discussed use them), or for the initiation of a downloaded DSP subroutine by the host. These are all directed through a number of reserved vectors within the DSP interrupt structure. There are however, two vectors which Atari have set aside for use by host programs (which by their nature are usually transient) as opposed to host subroutines (which usually remain resident). The relevant call is made by referring to the actual vector number, the hexadecimal value of $13 or $14. As far as the DSP program is concerned, the commands from the host can not be made until the relevant code has first been installed in the vectors within the DSP. This it does by placing the relevant `JMP` instruction at locations P:26 and P:28, which must point to the code somewhere within the body of the program. Finally, each time a program is downloaded into the DSP, the DSP vector table is overlaid by the Falcon's system vector table. All vectors may be overwritten except those at $13 and $14.

Dsp_Unlock — Relinquish DSP lock

BASIC 2 SUB Dsp_Unlock()

Devpac 3 dsp_unlock.W
(stack bytes)

Lattice C void Dsp_Unlock(void);

Any host process which has managed to claim the DSP for its own use *must* call this routine at some stage. Any process which terminates itself without first calling `Dsp_Unlock` will prevent any other process from using the DSP, including any further incarnations of the current process which may be launched in the future.

Chapter 6
GEMDOS/MiNT

What is MiNT?

The Atari GEMDOS system has been extended, chiefly for the implementation of the new multi tasking kernel within GEM called MiNT. These new calls allow systems programmers access to the *low level* control of multi tasking applications.

The system calls

Dclosedir — Close directory read

BASIC 2 FUNCTION Dclosedir&(BYVAL dirhandle&)

Devpac 3 dirhandle.L, d_closedir.W
(stack 6 bytes)

Lattice C long Dclosedir(long dirhandle);

`Dclosedir` will close the directory whose handle (returned from `Dopendir`) is `dirhandle`. The return is 0 if successful or `EIHNDL` if `dirhandle` is not valid.

Dcntl — Directory control operations

BASIC 2 FUNCTION Dcntl&(BYVAL cmd%, BYVAL name&,
BYVAL arg&)

Devpac 3 arg.L, name.L, cmd.W, d_cntl.W
(stack 12 bytes)

Lattice C long Dcntl(int cmd, const char *name, long arg);

`Dcntl` performs a file system specific command, given by `cmd`, upon the file or directory specified by `name`.

The exact nature of the operation performed depends upon the file system on which `name` resides. The interpretation of the third argument `arg` depends upon the specific command. The only built-in file system which supports `Dcntl` operations is the device file system `U:\DEV`.

The value returned by the call depends on the specific operation requested and the file system involved; a description of the detailed use of this call is beyond the scope of this book. Generally, a 0 or positive return value should mean success, -1 represents a failure.

Dgetcwd — *Get current working directory*

BASIC 2 FUNCTION Dgetcwd&(BYVAL path&, BYVAL drv%, BYVAL size%)

Devpac 3 size.W, drv.W, path.L, d_getcwd.W
(stack 10 bytes)

Lattice C long Dgetcwd(char *path, int drv, int size);

This call returns with the current working directory for a given `drv` in the area pointed to by `path`. In effect this is a replacement for the GEMDOS call `Dgetpath`, however, the length of the return is limited by size. This prevents the returned string from being longer than that of the allotted buffer, unlike `Dgetpath` which has the nasty side effect of running off the end of the buffer into other storage space if the string is too long to fit into the buffer pointed to by `path`.

Dlock — *Lock BIOS device*

BASIC 2 FUNCTION Dlock&(BYVAL mode%, BYVAL drv%)

Devpac 3 drv.W, mode.W, d_lock.W
(stack 6 bytes)

Lattice C long Dlock(int mode, int drv);

`Dlock` is used to lock or unlock the BIOS device indicated by `drv`. No GEMDOS file operations are permitted on a locked drive. Thus, the `Dlock` call provides a mechanism for disk formatters or re-organisers to lock out other processes while low-level BIOS or XBIOS operations are in progress on the device. If `mode` is 1, the drive is locked; if it is 0 then the drive is unlocked and may be used again by other programs. If a process terminates while holding a lock on a drive, that drive is automatically unlocked. A lock operation followed immediately by an unlock is very similar to a media change, except that the lock operation will fail if there are open files that refer to the indicated drive.

The call will return 0 if the lock/unlock operation was successful, EACCDN if mode is 1 and either open files exist on the drive or another process is searching a directory on the drive, ELOCKED if mode is 1 and another process has locked the drive, ENSLOCK if mode is 0 and the drive was not locked by this process, EDRIVE if drv is not a valid BIOS device number.

Note that Dlock operates on BIOS devices, which may not always be in 1-1 correspondence with GEMDOS drive letters. For this reason, to lock GEMDOS drive A: (for example), the programmer should call Fxattr on the root directory of A: (Fxattr(0, "A:\", ...)) and then use the dev field of the structure returned in order to determine the BIOS device corresponding to the GEMDOS drive.

Dopendir — Open directory for reading

BASIC 2 FUNCTION Dopendir&(BYVAL name&, BYVAL flag%)

Devpac 3 flag.W, name.L, d_opendir.W
(stack 8 bytes)

Lattice C long Dopendir(const char *name, int flag);

Dopendir opens the directory whose name is pointed to by name for reading. A 32 bit directory handle is returned which may be passed to Dreaddir to actually read the directory. flag controls the way directory operations are performed. If flag is 1, then the directory is read in *compatibility* mode, if flag is 0 then directory operations are performed in *normal* mode. In compatibility mode, file systems act as if the Fsfirst and Fsnext functions were being used; in particular, if it is possible file names will be restricted to the DOS 8.3 convention, and will be in upper case. In normal mode, file systems do not attempt to restrict the range of names. Moreover, in this mode the Dreaddir system call will also return a file index number (similar to the Unix *inode* number) along with the file name. New programs should generally use normal mode where possible.

This call will return a 32 bit directory handle, on success. Note that this handle may be negative, but will never contain the pattern 0xFF in the upper byte, whereas all errors do contain this pattern in the upper byte. Alternatively, one of the following error codes may be returned; EPTHNF if name is not a valid directory, EACCDN if the directory is not accessible by this program or ENSMEM if the kernel is unable to allocate memory needed for the directory operations.

Dpathconf Get configurable pathname variables

BASIC 2 FUNCTION Dpathconf&(BYVAL name&, BYVAL mode%)

Devpac 3 mode.W, name.L, d_pathconf.W
(stack 8 bytes)

Lattice C long Dpathconf(const char *name, int mode);

`Dpathconf` returns information about various limits or capabilities of the file system containing the file named `name`. The variable `mode` controls which limit or capability is being queried, as follows:

Value	Value Returned
-1	return max. legal value for `n` in `Dpathconf(n)`
0	return internal limit on the number of open files
1	return max. number of links to a file
2	return max. length of a full path name
3	return max. length of an individual file name
4	return number of bytes that can be written atomically
5	return information about file name truncation
6	return information about case sensitivity

If any of these items are unlimited, then 0x7fffffffL is returned.

For `mode` 5, return information about file name truncation, the returned value has the following meaning:

Value	Description
0	File names are never truncated; if the file name in any system call affecting this directory exceeds the maximum length (returned by mode 3), then the error value ERANGE is returned from that system call.
1	File names are automatically truncated to the maximum length.
2	File names are truncated according to DOS rules, i.e. to a maximum 8 character base name and a maximum 3 character extension.

For `mode` 6, information about case sensitivity, the returned value has the following meaning:

Value	Description
0	File system is case sensitive.
1	File system is case insensitive, and file case information is not preserved (e.g. file names are always converted to upper case).

Dreaddir — *Read directory entry*

BASIC 2 FUNCTION Dreaddir&(BYVAL len%, BYVAL dirhandle&, BYVAL buf&)

Devpac 3 buf.L, dirhandle.L, len.W, d_readdir.W (stack 12 bytes)

Lattice C long Dreaddir(int len, long dirhandle, void *buf);

`Dreaddir` returns the next file in the directory whose handle (from the `Dopendir` system call) is `dirhandle`. The file's name and (optionally) a 4 byte index for the file are placed in the buffer pointed to by `buf`. The file index is omitted if the directory was opened in compatibility mode (see `Dopendir` for details); otherwise, it is placed first in the buffer, followed by the (NUL terminated) name. If two names have the same index, then they refer to the same file; the converse, however, is not true. `len` is the size of the buffer, in total; it should be large enough to hold the index (if any), the file name, and the trailing 0. Successive calls to `Dreaddir` will return all the names in the directory, one after another, unless the `Drewinddir` system call is used to restart the reading at the beginning of the directory.

The call will return 0 if successful, `ERANGE` if the buffer was not large enough to hold the index (if present) and name or `ENMFIL` if there are no more file names to be read from the directory.

Drewinddir — Rewind directory read

BASIC 2 FUNCTION Drewinddir&(BYVAL handle&)

Devpac 3 handle.L, d_rewinddir.W
(stack 6 bytes)

Lattice C long Drewinddir(long handle);

`Drewinddir` rewinds the open directory whose handle (returned from the `Dopendir` system call) is handle. After the `Drewinddir` call, the next call to `Dreaddir` will read the first file in the directory. The call will return 0 if successful, `EIHNDL` if handle does not refer to a valid open directory or `EINVFN` if the directory cannot be rewound (for example, because of the type of file system on which it is located).

Fchmod — Modify extended file attributes

BASIC 2 FUNCTION Fchmod&(BYVAL name&, BYVAL mode%)

Devpac 3 mode.W, name.L, f_chmod.W
(stack 8 bytes)

Lattice C long Fchmod(const char *name, int mode);

`Fchmod` changes the file access permissions for the file named `name`. The new access permissions are given in the word mode, which may be constructed by ORing together the following symbolic constants:

`S_IRUSR, S_IWUSR, S_IXUSR`

Read, write, and execute permission (respectively) for the owner of the file.

`S_IRGRP, S_IWGRP, S_IXGRP`

Read, write, and execute permission (respectively) for the file's group.

`S_IROTH, S_IWOTH, S_IXOTH`

Read, write, and execute permission for everybody else.

These constants are defined in the `XATTR` structure. Please also note that not all file systems support all of these bits; bits not supported by a file system will be ignored. Note also that "execute" permission for a directory means permission to search the directory for a file name or name component.

The call will return 0 on success, EACCDN if the calling process has an effective uid which differs from the owner of the file and which is not 0, EFILNF if the file is not found or EPTHNF if the path to the file is not found.

Fchown — Change file ownership

BASIC 2 FUNCTION Fchown&(BYVAL name&, BYVAL uid%, BYVAL gid%)

Devpac 3 gid.W, uid.W, name.L, f_chown.W
(stack 10 bytes)

Lattice C long Fchown(const char *name, int uid, int gid);

This call will change a file's user and group ownership to uid and gid respectively. These ownership's determine access rights to the file. Only a process with effective uid 0 or whose effective uid matches the user ownership of the file may make this call. In the latter case, the new uid must match the old one, and the calling process must also be a member of the group specified by gid.

This call will return 0 if successful, EACCDN if the calling process has an effective uid which differs from the owner of the file and which is not 0, EINVFN if the file system on which the file is located does not support a notion of ownership (this is true of the normal TOS file system), EFILNF if the file is not found or EPTHNF if the path to the file is not found.

Fcntl — File control

BASIC 2 FUNCTION Fcntl&(BYVAL fh%, BYVAL arg&, BYVAL cmd%)

Devpac 3 cmd.W, arg.L, fh.W, f_cntl.W
(stack 10 bytes)

Lattice C long Fcntl(int fh, long arg, int cmd);

This call performs various control operations on the open file with GEMDOS file handle fh. The specific command to perform is given by cmd; the possible commands are given by symbolic constants and are listed below. arg is an argument whose meaning depends on the command.

The following commands are applicable to any file descriptor:

F_DUPFD $0000	Return a duplicate for the file handle. The new (duplicate) handle will be an integer ≥ arg and <32. If no free handles exist in that range, ENHNDL will be returned. The Fdup(fh) system call is equivalent to Fcntl(fh, 6L, F_DUPFD)
F_GETFD $0001	Return the noinherit flag for the file descriptor. This flag is 0 if child processes started by Pexec should inherit this open file, and 1 if they should not. arg is ignored.
F_SETFD $0002	Set the noinherit flag for the file descriptor from the low order bit of arg. The default value of the flag is 0 for handles 0-5 (the "standard" handles) and 1 for other (non-standard) handles. Note that the noinherit flag applies only to this particular file descriptor; another descriptor obtained from fh by the Fdup system call or by use of the F_DUPFD option to Fcntl will have the noinherit flag set to the default. Also note that these defaults are not the same as for UNIX.
F_GETFL $0003	Returns the user-settable file descriptor flags. These flags are the same as the mode passed to Fopen, unless they have been further modified by another Fcntl operation. arg is ignored.
F_SETFL $0004	Set user-settable file descriptor flags from the argument arg. Only the user-settable bits in arg are considered; the settings of other bits are ignored, but should be 0 for future compatibility. Moreover, it is not possible to change a file's read-write mode or sharing modes with this call; attempts to do so will (silently) fail.
FSTAT $4600	arg points to an XATTR structure, which is filled in with the appropriate extended file attributes for the file to which fh refers just as though the Fxattr system call had been made on the file. For a more detailed description of the this structure, please refer to the Fxattr function later on in this section.
FIONREAD $4601	arg points to a 32 bit integer, into which is written the number of bytes that are currently available to be read from this descriptor; a read of this number of bytes or less should not cause the process to block (wait for more input). Note that for some files only an estimate can be provided, so the number is not always completely accurate.

FIONWRITE $4602	arg points to a 32 bit integer, into which is written the number of bytes that may be written to the indicated file descriptor without causing the process to block. Note that for some files only an estimate can be provided, so the number is not always completely accurate.

The following Fcntl modes are used for file locking and take a structure of type flock:

```
struct flock {
 short l_type;    /* type of lock */
#define F_RDLCK 0
#define F_WRLCK 1
#define F_UNLCK 3
 short l_whence; /* SEEK_SET, etc. */
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
 long l_start;    /* start of region */
 long l_len;      /* length of lock */
 short l_pid;     /* callers pid */
};
```

F_GETLK $0005	arg is a pointer to an flock structure. If a lock exists which would prevent this lock from being applied to the file, the existing lock is copied into the structure and the l_pid field is set to the process id of the locking process. Otherwise, l_type is set to F_UNLCK. If a conflicting lock is held by a process on a different machine on a network, then the l_pid field will be set to a value defined by the network file system. This value will be in the range 0x1000 to 0xFFFF, and will therefore not conflict with any process id since process id's must be less than 0x1000.
F_SETLK $0006	Set (if l_type is F_RDLCK or F_WRLCK) or clear (if l_type is F_UNLCK) an advisory lock on a file. If the file is a FIFO, the whole file must be locked or unlocked at once, i.e. l_whence, l_start, and l_len must be 0. If this lock would conflict with a lock held by another process, ELOCKED is returned. If an attempt is made to clear a non-existent lock, ENSLOCK is returned. More than one read lock may be placed on the same region of a file, but no write lock may overlap with any other sort of lock. If a process holds locks on a file, then the locks are automatically released whenever the process closes an open file handle referring to that file, or when the process terminates.

F_SETLKW $0007	Like F_SETLK, but if the lock requested would conflict with a lock held by another process, the calling process is suspended until all conflicting locks are released.

The following commands are valid for any terminal device, e.g. the console or a pseudo-terminal:

TIOCGETP $5400	Get terminal parameters. `arg` is a pointer to a block of memory of type `struct sgttyb`.
TIOCSETP $5401	Set terminal parameters from the `struct sgttyb` pointed to by `arg`.

```
struct sgttyb {
 char sg_ispeed;      /* reserved */
 char sg_ospeed;      /* reserved */
 char sg_erase;       /* erase character */
 char sg_kill;        /* line kill character */
 short sg_flags;      /* terminal control flags */
};
```

TIOCGETC $5402	Get terminal control characters. `arg` is a pointer to a `struct tchars`.
TIOCSETC $5403	Set terminal control characters from the `struct tchars` pointed to by `arg`. Setting any character to the value 0 causes the corresponding function to become unavailable.

```
struct tchars {
 char t_intrc;        /* raises SIGINT */
 char t_quitc;        /* raises SIGQUIT */
 char t_startc;       /* starts terminal output */
 char t_stopc;        /* stops terminal output */
 char t_eofc;         /* marks end of file */
 char t_brkc;         /* marks end of line */
};
```

TIOCSLTC $5405	Set extended terminal control characters from the `struct ltchars` pointed to by `arg`. Setting any of the characters to 0 causes the corresponding function to become unavailable.
TIOCGLTC $5404	Get extended terminal control characters, and put them in the structure pointed to by `arg`:

```
struct ltchars {
 char t_suspc;        /* raises SIGTSTP now */
 char t_dsuspc;       /* raises SIGTSTP when read */
 char t_rprntc;       /* redraws the input line */
 char t_flushc;       /* flushes output */
 char t_werasc;       /* erases a word */
 char t_lnextc;       /* quotes a character */
};
```

TIOCSWINSZ $540C	arg has type `struct winsize *`. This structure contains the sizes for the current window. Note that the kernel maintains the information but does not act upon it in any way; it is up to window managers to perform whatever physical changes are necessary to alter the window size, and to raise the SIGWINCH signal if necessary.
TIOCGWINSZ $540B	arg has type `struct winsize *`. The current window size for this window is placed in the structure pointed to by arg.

If any fields in the structure are 0, this means that the corresponding value is unknown.

```
struct winsize {
 short ws_row;        /* # of rows of text in window*/
 short ws_col;        /* # of columns of text */
 short ws_xpixel;     /* width of window in pixels */
 short ws_ypixel;     /* height of window in pixels */
};
```

TIOCGPGRP $5406	arg is a long word pointer ; the process group for the terminal is placed into the long pointed to by it.
TIOCSPGRP $5407	arg has type `long *`; the process group for the terminal is set from the long pointed to by it. Processes in any other process group will be sent job control signals if they attempt input or output to the terminal.
TIOCSTART $540A	Restart output to the terminal (as though the user typed control-Q) if it was stopped by a control-S or TIOCSTOP command. arg is ignored.
TIOCSTOP $5409	Stop output to the terminal (as though the user typed control-S). arg is ignored.

TIOCGXKEY $540D	Get the definition of a function or cursor key. arg is a pointer to a `struct xkey`.
TIOCSXKEY $540E	Set the definition of a function or cursor key. `arg` is a pointer to a structure of type `struct xkey`.

```
struct xkey {
 short xk_num;        /* function key number */
 char xk_def[8];      /* associated string */
};
```

The string currently associated with the indicated key is in `xk_def`; this string is always null-terminated The `xk_num` field is the number of the desired key:

xk_num	Key
0-9	F1-F10
10-19	F11-F20 (shift F1-shift F10)
20	cursor up
21	cursor down
22	cursor right
23	cursor left
24	help
25	undo
26	insert
27	clr/home
28	shift+cursor up
29	shift+cursor down
30	shift+cursor right
31	shift+cursor left

`TIOCIBAUD` and `TIOCOBAUD` are used to select the input and output speeds for terminals. For these calls `arg` is a pointer to a long which the new rate is read from and the old rate written to. If the value pointed to by `arg` on entry is less than zero the rate is merely read, if the rate is zero DTR is dropped on those terminals which support it. If the terminal does not support split rates both input and output rates are set.

TIOCIBAUD $5412	Get/Set the input rate for the terminal given by `fh`.
TIOCOBAUD $5413	Get/Set the output rate for the terminal given by `fh`.

TIOCCBRK $5414	Clear break condition on terminal; arg is ignored.
TIOCSBRK $5415	Assert break condition on terminal; arg is ignored.

TIOCGFLAGS and TIOCSFLAGS are used to get and set, respectively, attributes of terminals such as number of data bits and number of stop bits.

TIOCGFLAGS $5416	Get the stop/data bits for the terminal referenced by fh into the 16 bit word value pointed to by arg.
TIOCSFLAGS $5417	Set the stop/data bits for the terminal referenced by fh from the 16 bit word value pointed to by arg.

The following definitions are used for the arg value:

Name	Value	Meaning
TF_1STOP	0x0001	One stop bit
TF_15STOP	0x0002	1.5 stop bits
TF_2STOP	0x0003	2 stop bits
TF_8BIT	0x0000	8 data bits
TF_7BIT	0x0004	7 data bits
TF_6BIT	0x0008	6 data bits
TF_5BIT	0x000C	5 data bits

The TCURS... family of Fcntl calls duplicate the standard XBIOS Cursconf functionality for generic terminals.

TCURSOFF $6300	Hide cursor; arg is ignored.
TCURSON $6301	Show cursor; arg is ignored.
TCURSBLINK $6302	Enable blinking; arg is ignored.
TCURSSTEADY $6303	Disable blinking; arg is ignored.
TCURSSRATE $6304	Set blink rate from the 16 bit word pointed to by arg.
TCURSGRATE $6305	Get blink rate into the 16 bit word pointed to by arg.

The following commands are valid only for processes opened as files (i.e. U:\PROC files):

PBASEADDR $5002	arg is a pointer to a 32 bit integer, into which the address of the process basepage for the process to which fh refers is written.
PPROCADDR $5001	arg is a pointer to a 32 bit integer, into which the address of the process control structure for the process is written.
PCTXTSIZE $5003	arg is a pointer to a 32 bit integer, into which the length of a process context structure is written. There are two of these structures located in memory just before the process control structure whose address is returned by the PPROCADDR command. The first is the current process context; the second is the saved context from the last system call.
PGETFLAGS $5005	arg is a pointer to a 32 bit integer, into which the process memory allocation flags are copied. These flags are the same ones found in the prgflags field of GEMDOS executable programs, or as the first parameter to Pexec mode 7.
PSETFLAGS $5004	arg is a pointer to a 32 bit integer, from which the process memory flags for the target process will be set. Note that only the low order 16 bits are actually used right now, and not all of these are valid. See the documentation for GEMDOS executable programs for details on the meanings of the flags.
PTRACEGFLAGS $5007	arg is a pointer to a 16 bit integer, into which the current trace flags of the target process are copied. If the process is not being traced, the flags will be 0.

PTRACESFLAGS $5007	`arg` is a pointer to a 16 bit integer, the bits of which determine how the target process will respond to signals. If bit #0 is set, the target process will respond to signals and other exceptions by stopping, and the process which set the flags will receive a SIGCHLD signal informing it of this fact; it may then use the `Pwaitpid` system call to retrieve information about why the process stopped, and may use `Fread` and `Fwrite` to interrogate and possibly change the state of the process before causing it to continue (see below). If bit #0 is clear, then all process tracing will cease, and the process will respond to signals in the normal way. All other bits are reserved and should be set to 0 for now. If some other process has already used PTRACESFLAGS to set process tracing for the target process, then the call will fail.
PTRACEGO $5008	Restarts a process that was being traced by the caller and which stopped because of a signal. `arg` is a pointer to a 16 bit integer which is either 0 (in which case all pending signals for the stopped process are cleared before it is restarted) or the number of a signal which is to be delivered to the process after it restarts. Typically, this will be the same as the signal that stopped it.
PTRACESTEP $500A	Like PTRACEGO, except that the trace bit will be set in the status register of the restarted process; thus, a SIGTRAP signal will be generated in that process after 1 user instruction has been executed. Note that it is not possible to trace processes that are executing in the kernel; if the process was stopped while executing in the kernel the trace bit will be set only when the process returns from the kernel.
PTRACEFLOW $5009	Again, this is similar to PTRACEGO except that it will run code until a change in the flow of instructions occurs. This is valid on 68020 processors and above. Any attempt to use this function on a 68000, for example, will return an error.

The following commands are valid only for files which represent shared memory:

SHMSETBLK $4D01	arg is a pointer to a block of memory previously allocated by Mxalloc. The memory will be offered for sharing under the name of the file represented by fh (which must be a file in the U:\SHM subdirectory).
SHMGETBLK $4D00	arg must be 0, for future compatibility. Returns the address of the block of memory previously associated with the file via SHMSETBLK, or a NULL pointer if an error occurs. Note that different processes may see the shared memory block at different addresses in their address spaces. Therefore, the shared memory block should not contain any absolute pointers to data.

The call will return 0 or a positive number if successful (for most commands; but see the specific descriptions above), EIHNDL if fh is not a valid GEMDOS open handle and EINVFN if the specified command is not valid for this file handle. Some other (long) negative error number may be generated if an error occurs and some of the different commands may recognise different possible errors.

Fgetchar — *Get character from file*

BASIC 2 FUNCTION Fgetchar&(BYVAL fh%, BYVAL mode%)

Devpac 3 mode.W, fh.W, f_getchar.W
(stack 6 bytes)

Lattice C long Fgetchar(int fh, int mode);

Fgetchar reads a character from the open file whose handle is fh. The parameter mode has an effect only if the open file is a terminal or pseudo-terminal, in which case the bits of mode have the following meanings:

0x0001 Cooked mode; special control characters (control-C and control-Z) are checked for and interpreted if found (they cause SIGINT and SIGTSTP, respectively, to be raised); also, flow control with control-S and control-Q is activated.

0x0002 Echo mode; characters read are echoed back to the terminal.

The ASCII value of the character read is put in the low byte of the long word that is returned. If the file is a terminal or pseudo-terminal, the scan code of the character pressed and (possibly) the shift key status are also returned in the long word, just as with the BIOS `Bconin` system call. The function returns the character read, if successful. 0x0000FF1A if end of file is detected. `EIHNDL` if `fh` is not a valid handle for an open file.

Finstat — *Get file input status*

BASIC 2 FUNCTION Finstat&(BYVAL fh%)

Devpac 3 fh.W, f_instat.W
(stack 4 bytes)

Lattice C long Finstat(int fh);

`Finstat` returns the number of bytes of input waiting on the file whose GEMDOS handle is `fh`, or 0 if no input is available on that handle. The return will be 0 or a positive number if successful It may also return `EIHNDL` if `fh` is not a valid handle for an open file.

Flink — *Create 'hard' link*

BASIC 2 FUNCTION Flink&(BYVAL oldname&,
BYVAL newname&)

Devpac 3 newname.L, oldname.L, f_link.W
(stack 10 bytes)

Lattice C long Flink(const char *oldname, const char *newname);

`Flink` creates a new name (a "hard link") for the file currently named `oldname`. If the `Flink` call is successful, then after the call the file may be referred to by either name, and a call to `Fdelete` on either name will not affect access to the file through the other name. `oldname` and `newname` must both refer to files on the same physical device. Note also that not all file systems allow links.

This call will return 0 if successful, `EXDEV` if `oldname` and `newname` refer to files on different physical devices, `EINVFN` if the file system does not allow hard links or `EFILNF` if the file named `oldname` does not currently exist.

Fmidipipe — Manipulate MIDI file handles

BASIC 2 FUNCTION Fmidipipe&(BYVAL pid%, BYVAL in%, BYVAL out%)

Devpac 3 out.W, in.W, pid.W, f_midipipe.W
(stack 8 bytes)

Lattice C long Fmidipipe(int pid, int in, int out);

`Fmidipipe` changes the MIDI input and output file handles (GEMDOS file handles -4 and -5 respectively) for process `pid`. `in` is the GEMDOS handle (for the calling process) which will become the MIDI input for the receiving process, and `out` is the GEMDOS handle which is to become the MIDI output.

If `pid` is 0, then the call affects the current process; in this case, it is roughly equivalent to the sequence; `Fforce(-4, in);Fforce(-5, out)`.

The call will return a value of 0 on success, `EFILNF` if the indicated process is not found, `EIHNDL` if either in or out is not a valid open handle or `EACCDN` if in is not open for reading or if out is not open for writing.

Foutstat — Get file output status

BASIC 2 FUNCTION Foutstat&(BYVAL fh%)

Devpac 3 fh.W, f_outstat.W
(stack 4 bytes)

Lattice C long Foutstat(int fh);

`Foutstat` returns the number of bytes of output that may be written to the file whose GEMDOS handle is `fh`, without blocking. The function returns 0 or a positive number if successful, `EIHNDL` if `fh` is not a valid handle for an open file.

Fpipe — Create anonymous pipe

BASIC 2 FUNCTION Fpipe%(BYVAL usrh&)

Devpac 3 usrh.L, f_pipe.W
(stack 6 bytes)

Lattice C int Fpipe(short usrh[2]);

`Fpipe` creates a pipe that may be used for interprocess communication. If it is successful, two GEMDOS handles are returned in `usrh[0]` and `usrh[1]`. `usrh[0]` will contain a handle for the read end of the pipe, (opened for reading only), and `usrh[1]` will contain a handle for the write end of the pipe (opened for writing only). The created pipe has the name `sys$pipe.xxx`, where `xxx` is a three digit integer. This call is typically used by shells; the shell redirects its standard input (or standard output) to the read (or write) end of the pipe using `Fdup` and `Fforce` before launching a child; the child will then read from (or write to) the pipe by default. The call will return 0 if successful, `ENHNDL` if there are not 2 free handles to allocate for the pipe, `ENSMEM` if there is not enough free memory to create the pipe, `EACCDN` if too many pipes already exist in the system. There cannot be more than 999 open pipes in the system at one time.

Fputchar — Put character to file

BASIC 2 FUNCTION Fputchar&(BYVAL fh%, BYVAL ch&, BYVAL mode%)

Devpac 3 mode.W, ch.L, fh.W, f_putchar.W
(stack 10 bytes)

Lattice C long Fputchar(int fh, long ch, int mode);

`Fputchar` outputs a character to the GEMDOS file whose handle is `fh`. The parameter mode has an effect only if the open file is a terminal or pseudo-terminal, in which case the bits of mode have the following meanings:

0x0001 Cooked mode; special control characters (control-C and control-Z) are checked for and interpreted if found (they cause `SIGINT` and `SIGTSTP`, respectively, to be raised); also, flow control with control-S and control-Q is activated.

If the file receiving output is a pseudo-terminal, then all 4 bytes of `ch` are recorded in the write, and may be retrieved by an `Fgetchar` call on the other side of the pseudo-terminal; this allows programs to pass simulated BIOS scan codes and shift key status through the pseudo-terminal. If the file receiving output is not a terminal, then only the low order byte of `ch` is written to the file. The function will return 4 (the number of bytes of data transferred) if the write was to a terminal, 1 if the write was not to a terminal and was successful, 0 if the bytes could not be output (for example, because of flow control) or `EIHNDL` if `fh` is not a valid handle for an open file. A (long) negative BIOS error code may appear if an error occurred during physical I/O.

Freadlink — *Read 'soft' link*

BASIC 2 FUNCTION Freadlink&(BYVAL bufsize%, BYVAL buf&, BYVAL name&)

Devpac 3 name.L, buf.L, bufsize.W, f_readlink.W
(stack 12 bytes)

Lattice C long Freadlink(int bufsiz, char *buf, const char *name);

`Freadlink` determines what file the symbolic link name points to, i.e. what the first argument to the `Fsymlink` call that created name was. This null terminated string is placed in the memory region pointed to by `buf`. The total size of this region is given by `bufsiz`; this must be enough to hold the terminating 0.

The call will return 0 on success, `ERANGE` if the symbolic link contents could not fit in `buf`, `EFILNF` if `name` is not found, `EACCDN` if `name` is not the name of a symbolic link or `EINVFN` if the file system containing the name does not support symbolic links.

Fselect — *Suspend process awaiting file I/O*

BASIC 2 FUNCTION Fselect%(BYVAL timeout%, BYVAL rfds&, BYVAL wfds&, 0&)

Devpac 3 $0.L, wfds.L, rfds.L, timeout.W, f_select.W
(stack 16 bytes)

Lattice C int Fselect(int timeout, long *rfds, long *wfds, NULL);

`Fselect` checks two sets of open file descriptors and determines which have data ready to read, and/or which are ready to be written to. If none are ready yet, the process goes to sleep until some member of the sets are ready or until a specified amount of time has elapsed.

`rfds` points to a long word which represents a set of GEMDOS file descriptors; bit n of this long word is set if file descriptor n is to be checked for input data. An empty set may optionally be represented by a `NULL` pointer instead of a pointer to 0. Similarly, `wfds` points to a 32 bit long word which indicates which file descriptors are to be checked for output status. When `Fselect` returns, the old values pointed to by `rfds` and `wfds` (if non-`NULL`) are overwritten by new long words indicating which file descriptors are actually ready for reading or writing; these will always form subsets of the file descriptors originally specified as being of interest.

`timeout` is a 16 bit unsigned integer specifying a maximum number of milliseconds to wait before returning; if this number is 0, no maximum is set and the call will block until one of the file descriptors specified is ready for reading or writing, as appropriate. Thus, `Fselect(0, 0, 0, NULL)` will block forever, whereas `Fselect(1, 0, 0, NULL)` will pause for 1 millisecond.

The final argument, a long word, must always be 0 (it is reserved for future enhancements).

The call will return the sum of the numbers of bits set in the long words pointed to by `rfds` and `wfds`. This will be 0 if the timeout expires without any of the specified file descriptors becoming ready for reading or writing, as appropriate, and non zero otherwise. The error code `EIHNDL` may be generated if any handle specified by the long words pointed to by `rfds` or `wfds` is not a valid (open) GEMDOS handle.

Fsymlink — *Create 'soft' link*

BASIC 2 FUNCTION Fsymlink&(BYVAL oldname&, BYVAL newname&)

Devpac 3 newname.L, oldname.L, f_symlink.W
(stack 10 bytes)

Lattice C long Fsymlink(const char *oldname, const char *newname);

`Fsymlink` creates a new symbolic link (a "soft link") for the file currently named `oldname`. If the `Fsymlink` call is successful, then after the call the file may be referred to by either name. A call to `Fdelete` on the new name will not affect the existence of the file, just of the symbolic link. A call to `Fdelete` with the name `oldname` will actually delete the file, and future references with `newname` will fail.

Unlike hard links, symbolic links may be made between files on different devices or even different types of file systems. The call will return 0 if successful, EINVFN if the file system does not allow symbolic links or another appropriate error code if the new symbolic link cannot be created.

Fxattr — *Obtain extended file attributes*

BASIC 2 FUNCTION Fxattr&(BYVAL flag%, BYVAL name&, BYVAL xattr&)

Devpac 3 xattr.L, name.L, flag.W, f_xattr.W
(stack 12 bytes)

Lattice C long Fxattr(int flag, const char *name, void *xattr);

`Fxattr` gets file attributes for the file named `name` and stores them in the structure pointed to by `xattr`. The following is the definition for the XATTR structure. Please note that this segment of C listing shows the numeric constants expressed in *octal*:

```
typedef struct xattr {
 unsigned short mode;              /* file types */
 #define S_IFMT          0170000  /* file type mask */
 #define S_IFCHR         0020000  /* BIOS file */
 #define S_IFDIR         0040000  /* directory file */
 #define S_IFREG         0100000  /* regular file */
 #define S_IFIFO         0120000  /* FIFO */
 #define S_IMEM          0140000  /* memory/process */
 #define S_IFLNK         0160000  /* symbolic link */

/* special bits: setuid, setgid, sticky bit */
 #define S_ISUID         04000     /* set user ID */
 #define S_ISGID         02000     /* set group ID */
 #define S_ISVTX         01000     /*retain text
                                   segment */
/* file access modes for user, group, and other*/
 #define S_IRUSR         0400
 #define S_IWUSR         0200
 #define S_IXUSR         0100
 #define S_IRGRP         0040
 #define S_IWGRP         0020
 #define S_IXGRP         0010
 #define S_IROTH         0004
 #define S_IWOTH         0002
 #define S_IXOTH         0001
```

```
 long index;
 unsigned short dev;
 unsigned short reserved1;
 unsigned short nlink;
 unsigned short uid;
 unsigned short gid;
 long size;
 long blksize, nblocks;
 short mtime, mdate;
 short atime, adate;
 short ctime, cdate;
 short attr;
 short reserved2;
 long reserved3[2];
} XATTR;
```

This structure contains the following fields of interest:

`unsigned short mode`	This field gives the file type and access permissions: (`mode & S_IFMT`) gives the file type (one of `S_IFCHR`, `S_IFDIR`, `S_IFREG`, `S_IFIFO`, `S_IMEM`, or `S_IFLNK`); (`mode & ~S_IFMT`) gives the file access mode according to the POSIX standard.
`long index`	An index for the file. Together with the `dev` field, this is intended to give a unique way of identifying the file. Note, however, that not all file systems are able to support this meaning, so it is best not to use this field unless absolutely necessary.
`unsigned short dev`	The device number for the file. This may be a BIOS device number as passed to the `Rwabs` function, or it may be a device number concocted by the file system to represent a remote device.
`unsigned short nlink`	Number of hard links to the file. Normally this field will be 1.
`unsigned short uid`	The user id of the owner of the file.
`unsigned short gid`	The group id of the owner of the file.
`long size`	The length of the file, in bytes.
`long blksize`	The size of blocks on this file system.

`long nblocks`	The number of physical blocks occupied by the file on the disk; this count includes any blocks that have been reserved for the file but do not yet have data in them, and any blocks that the file system uses internally to keep track of file data (e.g. Unix indirect blocks).
`short mtime`	The time of the last modification to the file, in standard GEMDOS format.
`short mdate`	The date of the last modification, in standard GEMDOS format.
`short atime, adate`	The time and date of the last access to the file, in GEMDOS format. Filesystems that do not keep this time will return the values given in `mtime` and `mdate` for these fields as well.
`short ctime, cdate`	The time and date of the file's creation, in GEMDOS format. Filesystems that do not keep this time will return the values given in `mtime` and `mdate` for these fields as well.
`short attr`	The standard TOS attributes for the file, as returned by `Fattrib` and/or `Fsfirst`.

The `flag` parameter controls whether or not symbolic links should be followed. If it is 0, then symbolic links are followed (like the Unix `stat` function). If `flag` is 1, then links are not followed and the information returned is for the symbolic link itself (if the named file is a symbolic link); this behaviour is like that of the Unix `lstat` system call.

This call will return 0 on success, `EFILNF` if the file is not found or `EPTHNF` if the path to the file is not found.

Pause — Suspend process awaiting signal

BASIC 2 SUB Pause()

Devpac 3 p_pause.W
(stack 2 bytes)

Lattice C void Pause(void);

`Pause` causes the calling process to go to sleep until a signal that is not being ignored or masked is received. If a signal handler has been established for that signal with the `Psignal` system call, then the handler is invoked before `Pause` returns; if the handler does a `longjmp` to a different point in the program, if it exits the program, or if the signal handler was set to `SIG_DFL` and the default action for the signal is to terminate the process, then `Pause` will never return.

Pdomain — Get/Set current process domain

BASIC 2 FUNCTION Pdomain%(BYVAL dom%)

Devpac 3 dom.W, p_domain.W
(stack 4 bytes)

Lattice C int Pdomain(int dom);

`Pdomain` gets or sets the process execution domain. This is a number which controls the behaviour of a process. The default domain is 0, which is the TOS compatibility domain and in which all system calls behave exactly as they do under TOS. Domain 1 is the MiNT domain; in this domain, the behaviour of the `Fread` and `Fwrite` system calls when applied to terminals are controlled by the current terminal settings as established by the `Fcntl` system call. Moreover, file names returned from `Fsfirst` and `Fsnext` may be treated differently; MiNT domain processes are expected to be able to deal with file names that are not standard 8 character name + 3 character extension, all upper case, DOS file names.

If `dom` is greater than or equal to 0, the process domain is set to its value. Note that only domains 0 and 1 are currently defined, and the result of using a different (positive) number for `dom` is unpredictable. If `dom` is negative, no change is made to the process domain.

This call will return the process domain at the time of the `Pdomain` call (i.e. before any change).

Pfork — Clone current process

BASIC 2 FUNCTION Pfork%()

Devpac 3 p_fork.W
(stack 2 bytes)

Lattice C int Pfork(void);

`Pfork` creates a copy of the current process. The child (the new process created) inherits a copy of the parent's address space, not the parent's original memory, and so changes to variables in the child do not affect the parent in any way. The new process begins execution with an apparent return from the `Pfork` call. The call will return 0 if it was in the child, the new process id (a positive number), if it was in the parent. An ENSMEM may be generated if there was not enough memory to create the new process.

Pgetpgrp — Get process group of current process

BASIC 2 FUNCTION Pgetpgrp%()

Devpac 3 p_getpgrp.W
(stack 2 bytes)

Lattice C int Pgetpgrp(void);

`Pgetpgrp` returns the process group number of the currently running process. Process groups are commonly used for job control and other signalling purposes; processes that share the same process group are assumed to be closely related, and are usually stopped all together rather than one at a time.

Pgetpid — Get current process ID

BASIC 2 FUNCTION Pgetpid%()

Devpac 3 p_getpid.W
(stack 2 bytes)

Lattice C int Pgetpid(void);

This returns the process id of the currently running process. This is a positive 16 bit integer which is unique among all processes currently in the system; the call is always successful.

Pgetppid — *Get parent process ID*

BASIC 2 FUNCTION Pgetppid%()

Devpac 3 p_getppid.W
(stack 2 bytes)

Lattice C int Pgetppid(void);

`Pgetppid` returns the process id of the parent of the currently running process. The process id is a positive 16 bit integer. The call is always successful. If the current process was started directly by the kernel, then `Pgetppid` will return 0.

Pgetuid, Pgetgid, Pgeteuid, Pgetegid — *Get real/effective user/group ID*

BASIC 2 FUNCTION Pgetuid%
FUNCTION Pgetgid%
FUNCTION Pgeteuid%
FUNCTION Pgetegid%

Devpac 3 p_getuid.W
p_getgid.W
p_geteuid.W
p_getegid.W
(stack 2 bytes)

Lattice C int Pgetuid(void);
int Pgetgid(void);
int Pgeteuid(void);
int Pgetegid(void);

`Pgetuid` returns the real user id of the currently running process. This is a number between 0 and 255 which determines the access permissions of the process, and which may be used in multi-user systems to distinguish different users of the system.

Similarly, `Pgetgid` returns the real group id of the currently running process; this will also be a number between 0 and 255. `Pgeteuid` and `Pgetegid` are similar to `Pgetuid` and `Pgetgid` respectively, except that they return the effective user or group id. This is normally the same as the real user or group id, except that if a program is run which has the `setuid` or `setgid` bit set, it will run with an effective user or group id equal to the owner of the program file. Access to files is based upon the effective user or group id, so the `setuid` (and `setgid`) mechanism allows users (in particular the super user) to grant permissions to other users. This mechanism also exists in the Unix operating system.

Pkill — *Send signal to process*

BASIC 2 FUNCTION Pkill%(BYVAL pid%, BYVAL sig%)

Devpac 3 sig.W, pid.W, p_kill.W
(stack 6 bytes)

Lattice C int Pkill(int pid, int sig);

`Pkill` sends the signal described by `sig` to one or more processes, as follows:

If `pid` is a positive number, then the signal is sent to the process with that process id. If `pid` is 0, the signal is sent to all members of the process group of the calling process (i.e. all processes which have the same process group number). This includes, of course, the calling process itself. If `pid` is less than 0, the signal is sent to all processes with process group number -`pid`.

The call will return 0 if successful. Note that if the current process is a recipient of the signal, the `Pkill` call may not return at all if the process is killed. ERANGE if `sig` is not a valid signal, EFILNF if `pid` > 0 and the indicated process has terminated or does not exist, or if `pid` < 0 and there are no processes in the corresponding process group. EACCDN if `pid` > 0, the sending process does not have an effective user id of 0, and the recipient process has a different user id from the sending process.

BASIC 2 FUNCTION Pmsg&(BYVAL mode%, BYVAL mboxid&, BYVAL msg&)

Devpac 3 msg.L, mboxid.L, mode.W, p_msg.W (stack 12 bytes)

Lattice C long Pmsg(int mode, long mboxid, void *msgptr);

This call is used for short messages and as a way to perform interprocess communication with little overhead. For more complicated messages or more general inter process communication, use FIFOs or pseudo-terminals.

`Pmsg` sends or receives a message to a specified message box. What sort of operation is performed depends on the bits in mode as follows:

Mode	Operation
0x0000	read
0x0001	write
0x0002	write, then read from `mboxid` 0xFFFFxxxx where xxxx is the process id of the current process
0x8000	OR with this bit to make the operation non-blocking.

The messages are five words long: two longs and a short, in that order. The values of the first two longs are totally up to the processes in question. The value of the short is the PID of the sender. On return from writes, the short is the PID of the process that read your message. On return from reads, it's the PID of the writer.

If the 0x8000 bit is set in the `mode`, and there is not a reader/writer for the `mboxid` already, this call returns -1. Otherwise, a read operation waits until a message is written and a write operation waits for a reader to receive the message.

In `mode` 2, the writer is declaring that it wants to wait for a reply to the message. What happens is that the reader gets put on the ready queue, but the writer is atomically turned into a reader on a mailbox whose `mboxid` is (0xFFFF0000 | pid). The idea is that this process will sleep until awoken at a later time by the process that read the message. The process reading the original request is guaranteed not to block when writing the reply.

This call will return 0 if successful, -1 if bit 0x8000 is set and the `Pmsg` call would have to block or `EINVFN` if mode is invalid.

Pnice — Adjust current process 'niceness'

BASIC 2 FUNCTION Pnice%(BYVAL delta%)

Devpac 3 delta.W, p_nice.W
(stack 4 bytes)

Lattice C int Pnice(int delta);

`Pnice` changes the priority of the base process by the amount `delta`. Higher levels of niceness correspond to decreased priority in scheduling, so positive values for `delta` cause the corresponding process to be scheduled less often, thus making the process 'nicer'. Conversely, negative values for `delta` cause the process priority to be increased. The adjusted process priority is returned as a 16 bit signed integer. 0 is the default priority; values greater than 0 are for higher priority processes (ones that are scheduled more often), and values less than 0 are lower priority processes.

Prenice — Adjust arbitrary process 'niceness'

BASIC 2 FUNCTION Prenice&(BYVAL pid%, BYVAL delta%)

Devpac 3 delta.W, pid.W, p_renice.W
(stack 6 bytes)

Lattice C long Prenice(int pid, int delta);

`Prenice` changes the base process niceness for the process number `pid`, by the amount `delta`. Higher levels of niceness correspond to decreased priority in scheduling, so positive values for `delta` cause the corresponding process to be scheduled less often. Conversely, negative values for `delta` cause the process priority to be increased.

The call will return the current priority for the process, if successful. This is a 16 bit signed quantity. The default priority is 0; higher priority processes have larger priority values, lower priority ones have smaller values. Alternatively, `EFILNF` will indicate that the specified process does not exist (note that since this is a 32 bit negative number it can be distinguished from the *word* negative numbers returned for low priority processes) or `EACCDN` if the process has a different user id.

Prusage — Obtain resource usage information

BASIC 2 SUB Prusage(BYVAL info&)

Devpac 3 info.L, p_rusage.W
(stack 6 bytes)

Lattice C void Prusage(long *info);

`Prusage` puts information on resources used by the current process into the memory pointed to by info, as follows:

`info[0]`	time spent by process in MiNT kernel
`info[1]`	time spent by process in its own code
`info[2]`	total kernel time spent by children of this process
`info[3]`	total user code time spent by children of this process
`info[4]`	memory allocated to this process (in bytes).
`info[5]`	reserved for future use.
`info[6]`	reserved for future use.
`info[7]`	reserved for future use.

Please note that all times returned as a result of this call are measured in milliseconds.

Psemaphore — Use uncounted semaphores

BASIC 2 FUNCTION Psemaphore&(BYVAL mode%, BYVAL id&, BYVAL timeout&)

Devpac 3 timeout.L, id.L, mode.W, p_semaphore.W
(stack 12 bytes)

Lattice C long Psemaphore(int mode, long id, long timeout);

`Psemaphore` is a call that implements uncounted semaphores. A semaphore is used for mutual exclusion: only one process at a time may own a given semaphore. For example, a semaphore may be used to protect access to data structures which are in shared memory and which are used by multiple threads in a process: before using the memory a thread must gain ownership of the guarding semaphore, and when finished the thread must release the semaphore. The semaphore would be created during initialisation and destroyed during shutdown.

Semaphores are identified by an ID, which is an arbitrary longword. This is the semaphore's name. The ID used to create the semaphore is the name of that semaphore from then on. When using semaphores, you should strive to use a longword that is unique. Using four ASCII characters which spell out something is common: 0x4b4f444f ("MODM") for instance might be the id of a semaphore that controls access to a modem. (Actually, this would be a poor choice, since there can be more than one modem in a system and this semaphore ID isn't flexible enough to handle that. "MDM1" might be better.) Please note that semaphore `id`'s beginning with 0x5f (the underscore character) are reserved for operating system use.

The `timeout` argument is only used in mode 2. It is ignored in other modes. A `timeout` of zero means return immediately. A value of -1 means forever - that is, never time out. Other values are a number of milliseconds to wait for the semaphore before timing out. The `mode` argument is used to tell what operation the caller desires:

Mode	Action
0	Create and "get" a semaphore with the given ID.
1	Destroy the indicated semaphore. The caller must own the semaphore prior to making this call.
2	Request ownership of the semaphore. Blocks until the semaphore is free or until the timeout expires. See below.
3	Release the semaphore. The caller must own the semaphore prior to making this call.

This call may return any of the following codes; 0 = Success, `ERROR` = A request for a semaphore which the caller already owns, `ERANGE` = The semaphore does not exist, `EACCDN` = Failure. The semaphore already exists (mode 0), you don't own it (modes 1 and 3), or the request timed out (mode 2).

Psetlimit — Set process resource limit

BASIC 2 FUNCTION Psetlimit&(BYVAL lim%, BYVAL value&)

Devpac 3 value.L, lim.W, p_setlimit.W
(stack 8 bytes)

Lattice C long Psetlimit(int lim, long value);

`Psetlimit` gets or sets a resource limit for a process. The limit which is affected is governed by the value of `lim`, as follows:

1	get/set maximum CPU time for process (milliseconds)
2	get/set total maximum memory allowed for process
3	get/set limit on Malloc'd memory for process.

If the data passed in value is negative, then the limit is unchanged; if value is 0, the corresponding resource is unlimited; otherwise, the resource limit is set to value.

Setting the maximum memory limit means the process is not allowed to grow bigger than that size overall. Setting the maximum Malloc'd limit means that the process may allocate no more than that much memory. The difference is that the latter limit applies above and beyond the text+data+bss size of the process.

Using `Psetlimit` sets the corresponding limit for both the process and any children it creates thereafter. Note that the limits apply to each process individually; setting the child CPU limit value to 1000 and then using `Pfork` to create three children results in each of those children getting a CPU limit value of one second. They do not have a collective or sum total limit of one second.

There is no restriction on increasing a limit. Any process may set any of its limits or its childrens limits to a value greater than its current limit, or even to zero (unlimited).

Memory limits do not apply during execution of `Pexec`; that is, if a process is limited to (say) 256KB of memory, it can still exec a child which uses more. Memory limits are not retroactive: if a process owns 256KB of memory and then calls `Psetlimit` to restrict itself to 128KB, it will not be terminated, but no `Malloc` calls will succeed until its size drops below 128KB.

CPU limits are retroactive, however: if a process has used three CPU seconds and calls `Psetlimit` to restrict itself to one second, it will immediately receive `SIGXCPU` and terminate.

As a result of the call the returned value will represent the old limit, where a value of 0 will mean that there was no limit.

Psetpgrp — *Set process group of current process*

BASIC 2 FUNCTION Psetpgrp%(BYVAL pid%, BYVAL newgrp%)

Devpac 3 newgrp.W, pid.W, p_setpgrp.W
(stack 6 bytes)

Lattice C int Psetpgrp(int pid, int newgrp);

`Psetpgrp` sets the process group of the process with process id `pid` to the number `newgrp`. The process must have the same user id as the current process, or must be a child of that process. If `pid` is 0, the process group of the current process is set. If `newgrp` is 0, then the process group is set equal to the id of the process. this call returns the new process group number if successful, `EFILNF` if process `pid` does not exist or `EACCDN` if the process has a different user id and is not a child of the calling process.

Psetuid, Psetgid — *Set real user/group ID*

BASIC 2 FUNCTION Psetuid%(BYVAL uid%)
FUNCTION Psetgid%(BYVAL gid%)

Devpac 3 uid.W, p_setuid.W
gid.W, p_setgid.W
(stack 2 bytes)

Lattice C int Psetuid(int uid);
int Psetgid(int gid);

`Psetuid` sets the user id of the current process to `uid`, which must be a number between 0 and 255 inclusive. This function call will fail if the user id is not already 0, so once a process' user id is set, it is fixed. `Psetuid` returns `uid`, if the call is successful, and `EACCDN` if the process does not have the authority to change its own user id (i.e. if its effective user id is not 0 at the time of the call).

Similarly, `Psetgid` sets the group id of the current process to gid, which again must be between 0 and 255. The new group id is returned if successful, and `EACCDN` is returned if permission to change groups is denied.

Psigaction Install POSIX.1 style signal handler

BASIC 2 FUNCTION Psigaction&(BYVAL sig%, BYVAL act&, BYVAL oact&)

Devpac 3 oact.L, act.L, sig.W, p_sigaction.W
(stack 12 bytes)

Lattice C long Psigaction(int sig, const void *act, void *oact);

`Psigaction` changes the handling of the signal indicated by `sig` (which must be between 1 and 31; as defined in the appendix MiNT signals). If `act` is non-zero, then it is assumed to point to a structure describing the signal handling behaviour. This structure has the following members:

```
struct sigaction {
 void (*__stdargs sa_handler)(long);
 long sa_mask;
 short sa_flags;
}
```

If `sa_handler` is `SIG_DFL`, then the default action for the signal will occur when the signal is delivered to the process.

If `sa_handler` is `SIG_IGN`, then the signal will be ignored by the process, and delivery of the signal will have no noticeable effect (in particular, the signal will not interrupt the `Pause` or `Psigpause` system calls). If the signal is pending at the time of the `Psignal` call, it is discarded.

If `sa_handler` is some other value, it is assumed to be the address of a user function that will be called when the signal is delivered to the process. The user function is called with a single *long* argument on the stack, which is the number of the signal being delivered (this is done so that processes may use the same handler for a number of different signals). While the signal is being handled, it is blocked from delivery; thus, signal handling is "reliable" (unlike Version 7 and early System V Unix implementations, in which delivery of a second signal while it was being handled could kill the process). The set of signals specified in `sa_mask` are also blocked from delivery while the signal handler is executing. Note that, unlike in some versions of Unix, the signal handling is not reset to the default action before the handler is called; it remains set to the given signal handler.

The signal handler must either return (via a normal 680x0 RTS instruction) or call the Psigreturn system call to indicate when signal handling is complete; in both cases, the signal will be unblocked. Psigreturn also performs some internal clean-up of the kernel stack that is necessary if the signal handler is not planning to return (for example, if the C longjmp() function is to be used to continue execution at another point in the program).

Signal handlers may make any GEMDOS, BIOS, or XBIOS system calls freely. GEM AES and VDI calls should not be made in a signal handler.

The sa_flags field specifies additional, signal-specific signal handling behaviour. If sig is SIGCHLD, and the SA_NOCLDSTOP bit is set in sa_flags, then the SIGCHLD signal is sent to this process only when one of its children terminates (and not when a child is suspended by a job control signal).

The oact argument to Psigaction, if non-zero, specifies a structure that will be set to reflect the signal handling for sig that was current at the time of the Psigaction system call.

Note that calling Psigaction to change the behaviour of a signal has the side effect of unmasking that signal, so that delivery is possible. This is done so that processes may, while handling a signal, reset the behaviour and send themselves another instance of the signal, for example in order to suspend themselves while handling a job control signal. Signal handling is preserved across Pfork and Pvfork calls. Signals that are ignored by the parent are also ignored by the child after a Pexec call; signals that were being caught for handling in a function are reset in the child to the default behaviour. The call returns 0 on success, ERANGE if sig is not a legal signal, EACCDN if the signal may not be caught by the user.

A special note to BASIC 2 users: It is not possible to write handling routines in BASIC, it is necessary to have written these using an assembler such as Devpac and then linked them into the code produced by BASIC at a later stage.

Psigblock, Psigsetmask — *Set signal mask*

BASIC 2 FUNCTION Psigblock&(BYVAL mask&)
FUNCTION Psigsetmask&(BYVAL mask&)

Devpac 3 mask.L, p_sigblock.W
mask.L, p_sigsetmask.W
(stack 6 bytes)

Lattice C long Psigblock(long mask);
long Psigsetmask(long mask);

`Psigblock` adds the set of signals defined by the variable mask to the set of signals which are blocked from delivery. Each bit of `mask` represents a signal; if bit N of mask is set, then signal number N is blocked. `Psigblock` returns the set of blocked signals as it was prior to the new signals being added to it; the old set can thus be restored with the `Psigsetmask` call.

`Psigsetmask` replaces the set of blocked signals with the set in `mask`; the bits of `mask` have the same meaning as they do for `Psigblock`, except that bits that are set to 0 will cause the corresponding signals to no longer be blocked. `Psigsetmask` returns the old set of blocked signals.

Note that certain signals (e.g. `SIGKILL`) cannot be blocked, and the kernel will (silently) clear the corresponding bits in `mask` before changing the blocked signal set.

Blocked signals remain blocked across `Pfork` and `Pvfork` calls. After a `Pexec` call, the child process will always start with an empty set of blocked signals, regardless of which signals were blocked by the parent.

Psignal — *Install signal handler*

BASIC 2 FUNCTION Psignal&(BYVAL sig%, BYVAL handler&)

Devpac 3 handler.L, sig.W, p_signal.W
(stack 6 bytes)

Lattice C void (*__stdargs Psignal(int sig,
void (*__stdargs handler)(long))(long);

`Psignal` changes the handling of the signal indicated by `sig` (which must be between 1 and 31 inclusive; these are defined in the appendix MiNT error codes). If `handler` is `SIG_DFL`, then the default action for the signal will occur when the signal is delivered to the process.

If `handler` is `SIG_IGN`, then the signal will be ignored by the process, and delivery of the signal will have no noticeable effect (in particular, the signal will not interrupt the `Pause` or `Psigpause` system calls). If the signal is pending at the time of the `Psignal` call, it is discarded.

If `handler` is some other value, it is assumed to be the address of a user function that will be called when the signal is delivered to the process. The user function is called with a single long argument on the stack, which is the number of the signal being delivered (this is done so that processes may use the same handler for a number of different signals). While the signal is being handled, it is blocked from delivery; thus, signal handling is "reliable" (unlike Version 7 and early System V Unix implementations, in which delivery of a second signal while it was being handled could kill the process). Note that, unlike in some versions of Unix, the signal handling is not reset to the default action before the handler is called; it remains set to the given signal handler.

The signal handler must either return (via a normal 680x0 `RTS` instruction) or call the `Psigreturn` system call to indicate when signal handling is complete; in both cases, the signal will be unblocked. `Psigreturn` also performs some internal clean-up of the kernel stack that is necessary if the signal handler is not planning to return (for example, if the C `longjmp()` function is to be used to continue execution at another point in the program).

Signal handlers may make any GEMDOS, BIOS, or XBIOS system calls freely; GEM AES and VDI calls however should not be mixed in a signal handler.

Please also note that calling `Psignal` to change behaviour of a signal has the side effect of unmasking that signal, so that delivery is possible. This is done so that processes may, while handling a signal, reset the behaviour and send themselves another instance of the signal, for example in order to suspend themselves while handling a job control signal. Signal handling is preserved across `Pfork` and `Pvfork` calls. Signals that are ignored by the parent are also ignored by the child after a `Pexec` call; signals that were being caught for handling in a function are reset in the child to the default behaviour. The call will return the old value of the signal handler on success; this will be either `SIG_DFL`, `SIG_IGN`, or a function address, `ERANGE` if `sig` is not a legal signal or `EACCDN` if the signal may not be caught by the user.

A special note to BASIC 2 users: It is not possible to write handling routines in BASIC, it is necessary to have written these using an assembler such as Devpac and then linked them into the code produced by BASIC at a later stage.

Psigpause — *Pause awaiting signal with mask*

BASIC 2 SUB Psigpause(BYVAL mask&)

Devpac 3 mask.L, p_sigpause.W
(stack 4 bytes)

Lattice C void Psigpause(long mask);

`Psigpause` sets a new signal mask and then causes the calling process to go to sleep until a signal that is not being ignored or masked is received. If a signal handler has been established for that signal with the `Psignal` system call, then the handler is invoked before `Psigpause` returns; if the handler does a `longjmp` to a different point in the program, if it exits the program, or if the signal handler was set to `SIG_DFL` and the default action for the signal is to terminate the process, then `Psigpause` will never return.

If `Psigpause` does return, then the signal mask is restored to that prior to the `Psigpause` system call, i.e. the new signal mask specified by `mask` is only temporary.

Psigpending — *Inquire pending signals*

BASIC 2 FUNCTION Psigpending&()

Devpac 3 p_sigpending.W
(stack 2 bytes)

Lattice C long Psigpending(void);

`Psigpending` returns a bit mask containing the signals that have been sent to the calling process but not yet delivered (probably because they have been blocked, either directly via `Psigblock` or `Psigsetmask`, or indirectly because of signal handling. The call will return with a long value in which bits will be set if (and only if) the relevant signal is pending.

Psigreturn — Prepare kernel for signal exit

BASIC 2 SUB Psigreturn()

Devpac 3 p_sigreturn.W
(stack 2 bytes)

Lattice C void Psigreturn(void);

`Psigreturn` is used to prepare to exit from a signal handler. This is done automatically by the kernel when a signal handler returns, so it is needed only before a program uses the C `longjmp` function (or some similar facility) to do a non-local jump. `Psigreturn` will fail (harmlessly) if no signal is being processed at the time it is called.

Pumask — Set process file creation mask

BASIC 2 FUNCTION Pumask%(BYVAL mode%)

Devpac 3 mode.W, p_umask.W
(stack 4 bytes)

Lattice C int Pumask(int mode);

`Pumask` changes the file and directory creation mask of the current process to the unsigned 16 bit quantity specified in `mode`. The old value of the creation mask is returned as a result of the call. Child processes inherit the new value for the mask.

When a new file is created with `Fcreate` or a new directory created with `Dcreate`, the initial access permissions (see `Fchmod` for a description of these) for the newly created file or directory are normally set so that all permissions are granted (except that execute permission is not normally granted for files). The creation mask set by `Pumask` determines which permissions are not to be granted by default. Thus, files created after a `Pumask(S_IWOTH | S_IWGRP | S_IXOTH)` call will be readable by anyone, but writeable only by the owner; moreover, directories created after this call would be searchable by the owner and members of the same group, but not by anyone else.

Pusrval — Get/Set user process value

BASIC 2 FUNCTION Pusrval&(BYVAL val&)

Devpac 3 val.L, p_usrval.W
(stack 6 bytes)

Lattice C long Pusrval(long val);

`Pusrval` may be used to set or retrieve the old process specific user value. This is a long word which is attached to the process, and is inherited by child processes. The use and meaning of the value is entirely up to applications; the kernel only records it. If `val` is -1, then no change is made to the user value; otherwise it is set to `val`. This call will return the old process specific user value.

Pvfork — Clone current process

BASIC 2 FUNCTION Pvfork%()

Devpac 3 p_vfork.W
(stack 2 bytes)

Lattice C int Pvfork(void);

`Pvfork` creates a copy of the current process. Both the child (the new process created) and the parent (the process which first made the call) share the same address space, i.e. any changes that the child makes to variables will also be noticed by the parent. The new process begins execution with an apparent return from the `Pvfork` call.

Because the two processes share the same address space, including the same stack, there could be many problems if they actually were running at the same time. Therefore, the parent process is suspended until the child process either exits or uses mode 200 of `Pexec` to overlay itself with a new program in a new address space.

This call will return 0 if in the child, the new process id (a positive number) if in the parent or ENSMEM if there is not enough memory to create the new process.

Pwaitpid, Pwait, Pwait3 Collect child exit codes

BASIC 2 FUNCTION Pwait&
FUNCTION Pwait3&(BYVAL flag%, BYVAL rusage&)
FUNCTION Pwaitpid&(BYVAL pid%, BYVAL flag%, BYVAL rusage&)

Devpac 3 p_wait.W
(stack 2 bytes)
rusage.L, flag.W, p_wait3.W
(stack 8 bytes)
rusage.L, flag.W, pid.W, p_waitpid.W
(stack 10 bytes)

Lattice C long Pwait(void);
long Pwait3(int flag, long *rusage);
long Pwaitpid(int pid, int flag, long *rusage);

`Pwaitpid` attempts to determine the exit code for certain stopped children or children that have terminated. `pid` determines which childrens exit status are of interest. If `pid` is -1, all children are of interest. If `pid` is less than -1, only children whose process group matches -`pid` are of interest. If `pid` is equal to 0, only children with the same process group ID as the caller are of interest. If `pid` is greater than 0, only the child with the given process ID is of interest.

If bit 1 of `flag` is set, then children that are stopped due to job control are reported; otherwise only children that have actually terminated are reported. A stopped process will be reported only once (unless it is re-started and stopped again); similarly a terminated process will be reported only once.

If the process does have children which are of interest to the caller (as specified by `pid`), but none are currently stopped or terminated and not yet waited for, then the behaviour of `Pwaitpid` is controlled by bit 0 of `flag`. If it is clear, the function will wait until some child is stopped or terminates; if it is set, the function will return immediately.

The `rusage` parameter, if non-zero, should point to two long words, into which information about the child's CPU time usage is placed, as follows:

`rusage[0]`	milliseconds spent by child in user space
`rusage[1]`	milliseconds spent by child in kernel space

If a child process is found which matches the `pid` specification given, its process id is placed in the upper 16 bits of the 32 bit value returned, and its exit status (as passed to `Pterm` or `Ptermres`, as determined implicitly by `Pterm0`, or as determined by the type of signal that stopped or killed the process) is placed in the lower 16-bits. If the process was stopped or terminated by signal n, then its exit status will be `(n<< 8) | x` where `x` is 127 if the process was stopped and 0 if the process was terminated. 0 if bit 0 of `flag` is set and the `Pwaitpid` system call would have otherwise blocked waiting for a child to exit or stop. Finally, `EFILNF` if no unwaited for children exist which are "of interest" as specified by `pid`.

The `Pwait3()` system call is equivalent to `Pwaitpid(-1, flag, rusage)`, and is provided for backward compatibility. The `Pwait()` system call is equivalent to `Pwaitpid(-1, 2, NULL)` and is provided only for compatibility with old applications; new applications should not use these system calls.

Salert — *Generate system alert message*

BASIC 2 SUB Salert(BYVAL msg&)

Devpac 3 msg.L, s_alert.W
(stack 6 bytes)

Lattice C void Salert(const char *msg);

`Salert` sends a warning or error message to the alert pipe, `U:\PIPE\ALERT`. The argument `msg` is a 0 terminated ASCII string containing the message to be sent. The message should not contain any carriage return, line feed, or escape characters; it should be a simple one line warning or error message which is to be brought to the user's attention. The `Salert` call takes whatever steps are necessary to format this string and send it to the user; the exact form of the output (or even whether the output is seen at all) depends on the system configuration.

Syield — Relinquish processor

BASIC 2 SUB Syield

Devpac 3 s_yield.W
(stack 2 bytes)

Lattice C void Syield(void);

`Syield` gives up control of the processor temporarily. Use of this call in a tight loop can consume a fair amount of processor time, for this reason it might be better to consider the use of `Fselect`, `Talarm` or `Pause` instead if possible.

Sysconf — Get configurable system variables

BASIC 2 FUNCTION Sysconf&(BYVAL n%)

Devpac 3 n.W, s_sysconf.W
(stack 4 bytes)

Lattice C long Sysconf(int n);

`Sysconf` returns information about various limits or capabilities of the currently running version of MiNT. The variable `n` controls which limit or capability is being queried, as follows:

n	Value Returned
-1	return max. legal value for `n` in `Sysconf(n)`
0	return max. number of memory regions per process
1	return max. length of `Pexec` command line string
2	return max. number of open files per process
3	return number of supplementary group IDs
4	return max. number of processes per user

If any of these items are unlimited, then 0x7fffffffL is returned.

Talarm — *Schedule alarm*

BASIC 2 FUNCTION Talarm&(BYVAL time&)

Devpac 3 time.L, t_alarm.W
(stack 6 bytes)

Lattice C long Talarm(long time);

This call will allow a process to set up an alarm which is to occur no sooner than `time` seconds in the future. Sometime later (which may actually be longer than the set value) a `SIGALRM` will be delivered to the process. Please note that unless a handler for `SIGALRM` has been established by means of the `Psignal` function, then the arrival of the `SIGALRM` will kill the process.

Once an alarm has been setup by the process and while that alarm is still pending, then the action of a further call will depend on the value set by time. If `time` is 0, then the previously scheduled alarm will be cancelled. If `time` is a negative number, then it will return the number of seconds left before the alarm is due. If the value passed in `time` is a positive value, then the alarm will be re-scheduled to return the `SIGALRM` in a further `time` seconds and the pending time of the old alarm will be returned as a result.

Important GEMDOS extensions

The following is a list of standard GEMDOS calls which have been extended for the new multi-tasking environment. In essence, they should work as documented else where in your language manuals but please note how the new extensions affect some of the calls.

Dfree — *Get free disk space*

BASIC 2 SUB Dfree(buf&(), BYVAL d%)

Devpac 3 d.W, buf.L, d_free.W
(stack 8 bytes)

Lattice C long Dfree(long *buf, int d);

In most normal circumstances this call would return the amount of free storage that is available on the currently selected device. The pseudo device `U:` under MiNT includes modifies this behaviour so that the call returns information based on the current path (when requesting `U:` based information).

For `U:\PIPE` the following information is returned:

`buf[0]`	maximum available pipes
`buf[1]`	total pipes available
`buf[2]`	size of pipe
`buf[3]`	1

For `U:\PROC` and `U:\SHM` the following information is returned:

`buf[0]`	number of free pages
`buf[1]`	total number of pages
`buf[2]`	page size
`buf[3]`	1

Note that the BASIC binding does not return errors.

Flock — *File locking primitive*

BASIC 2 FUNCTION Flock&(BYVAL handle%, BYVAL mode%, BYVAL start&, BYVAL length&)

Devpac 3 length.L, start.L, mode.L, handle.L, f_lock.W (stack 18 bytes)

Lattice C long Flock(int handle, int mode, long start, long length);

The `Flock` function is designed to lock a specified portion of an open file to prevent other processes from accessing and/or modifying that part of the file. This function is may be available for systems which have file locking extensions, indicated by the `_FLK` cookie, but without MiNT present; if MiNT is present the `Fcntl(F…LK…, …)` should be used in preference (although MiNT also installs an `_FLK` cookie indicating the presence of this call).

`handle` is the GEMDOS file handle of the open file, `mode` specifies if the portion of the file is being locked or unlocked:

Value	Meaning
0	Create a lock, starting at `start` and extending for `length` bytes.
1	Remove a previously set lock; `start` and `length` must match a previous lock.

The `start` parameter is the offset from the start of the file, in bytes, where the lock will begin. The `length` parameter is the length of the locked area, in bytes. Once a lock has been set using a certain handle, other file handles will not be able to read or write the locked area of the file. If there are outstanding locks when a file handle is closed, the behaviour is undefined.

Fopen — *Open a file*

BASIC 2 FUNCTION Fopen&(BYVAL name$, BYVAL mode%)

Devpac 3 mode.W, name.L, f_open.W
(stack 8 bytes)

Lattice C long Fopen(const char *name, int mode);

As in previous versions of GEMDOS it is possible to open a file for reading or writing. This value used to be between 0 and 2 but it has now been extended considerably to account for file locking and additional flags:

O_RDONLY	0x00	read from file only
O_WRONLY	0x01	write to file only
O_RDWR	0x02	read or write to file
O_APPEND	0x08	all writes go to end of file
O_COMPAT	0x00	compatibility mode
O_DENYRW	0x10	deny both read and write access
O_DENYW	0x20	deny write access to others
O_DENYR	0x30	deny read access to others
O_DENYNONE	0x40	don't deny any access to others
O_NOINHERIT	0x80	private file (not passed to child)
O_NDELAY	0x100	don't block for I/O on this file
O_CREAT	0x200	create file if it doesn't exist
O_TRUNC	0x400	truncate file to 0 bytes if it does exist
O_EXCL	0x800	fail open if file exists
O_GLOBAL	0x1000	for opening a global file

The extension flags list above are those relevant to MiNT; if you find that the `_FLK` cookie has been installed, but no `MiNT` cookie, only a subset of these are available: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_COMPAT`, `O_DENYRW`, `O_DENYW`, `O_DENYR`, `O_DENYNONE`, `O_NOINHERIT`.

Mxalloc Allocate memory from preferred pool

BASIC 2 FUNCTION Mxalloc&(BYVAL size&, BYVAL mode%)

Devpac 3 mode.W, size.L, m_xalloc.W
(stack 8 bytes)

Lattice C long Mxalloc(long size, int mode);

Request memory from the system. The first 2 bits (values 0 to 3) represent the preference of the memory to be returned. The next 2 bits (values 4, 8 and 9) select the protection of the memory from other applications.

Mode	Meaning
$0000	System RAM only
$0001	alternative RAM only
$0002	either, system RAM preferred
$0003	either, alternative preferred.

The protection bits are as follows:

Mode	Meaning
$0000	Default (from the program header flags)
$0010	Private
$0020	Global
$0030	Supervisor
$0040	World-readable

The high order bit number 14 ($4000) serves a special function. When set, this bit means "if the owner of this region terminates, don't free this region. Instead, let MiNT inherit it, so it'll never be freed."; this is a special mode meant for the OS only.

BASIC 2 FUNCTION Pexec&(BYVAL mode%, BYVAL path$, BYVAL tail$, BYVAL env$)

Devpac 3 env.L, tail.L, path.L, mode.W, p_exec.W (stack 16 bytes)

Lattice C long Pexec(int mode, const char *path, const char *tail, const char *env);

`Pexec` provides facilities for a program to create basepages, load programs and execute them.

`path` is a pointer a string giving the filename of the program to execute. If `path` does not specify a drive the current drive is used, similarly if no pathname is specified the current path is used. Note that any filename extension must be explicitly specified.

`tail` is a pointer to a length prefixed string, i.e. `tail[0]` contains the length of the string starting at `tail[1]`, the total length of the string (including the length byte) may not exceed 126 bytes. Note that when copying this string GEMDOS copies 126 bytes or up to a NUL character, which ever is first.

`env` contains a pointer to the environment to be passed to the child process. If this pointer is `NULL` then the child inherits a copy of the parents environment. GEMDOS obtains a block of memory using `Malloc` into which it copies the child processes environment.

The `mode` parameter determines what function the command performs. The following `mode` values are allowed:

Value	Meaning
0	Create a basepage, load program into the basepage, execute program returning program's termination code when the program completes.
3	Create a basepage and load program into it. The value returned is the address of the base page created.
4	Execute program already loaded. For this mode `path` and `env` are unused (pass `NULL` for these). `tail` holds the address of the program to execute. The value returned is the program termination code. Note that the TPA and environment are not freed after running the program.

5	Create a basepage. For this mode `path` is unused (pass `NULL` for this), `tail` and `env` have there normal meanings. The value returned is the address of the base page created.
6	Execute program already loaded. For this mode `path` and `env` are unused, and `tail` holds the address of the program to execute. The value returned is the program termination code. Unlike mode 4, the TPA and environment are freed after executing the child process. Note the warning below about this mode.
7	Create a basepage. For this mode `path` holds the program load flags as set in the executable file's load bits. `tail` and `env` have their normal meanings. The value returned is the address of the base page created. Note the warning below about this mode.
100 104 106	As mode 0, 4 and 6, but start program asynchronously. The value returned is the process ID of the new child.
200 204 206	As mode 0, 4 and 6 but overlay existing program (this is equivalent to the UNIX `exec` function). This function does not (normally) return.

If bit 15 ($8000) of `mode` is set process tracing for the new process is automatically enabled (see `Fcntl(PTRACESFLAGS, ...)`).

Chapter 7
AES Enhancements

This appendix will provide the reader with an overview of the extensions which have been made to the AES. New features have been added to support new graphical and control features such as popup menus and the new multitasking version of TOS known as MultiTOS. It is outside the scope of this manual to pursue an in-depth discussion of the new AES calls, but an attempt is made to explain the differences between the other language documentation provided with this package and that which has changed or been added to since publication.

An AES dilemma

For all intents and purposes, these extensions have been made in two major phases. First of all came the extensions for MultiTOS. These new features were made available from version 4.00 of the AES onwards. At this point a new version of the AES was split off for development purposes, the numbering of which started from version 3.30! This numbering system may seem a little strange but it does in fact make some sense since the changes made for version 3.30 were for a single tasking system, but they needed to be developed with future multi tasking compatibility in mind, since the multi tasking was already at the core of the code on which 3.30 was based, this could now be taken for granted.

Any application which uses the AES must check to see which version of the system it is running under. It must now go without saying that a single tasking application only needs to know if the version is 3.30 or above to be sure that the new features are available. A multi tasking application may not require the new pop up menus (for example) but must know that multi-tasking is available for it to make certain calls, for this reason an AES version of 4.00 or above will be fine!

What's in AES 3.30...

The key features of the AES from 3.30 are:

- Hierarchical menus
- Pop-up dialog menus
- Scrolling of information within pop-up menus and sub menus
- 3D controls on dialogs and windows
- Colour resource files which feature colour and animated icons

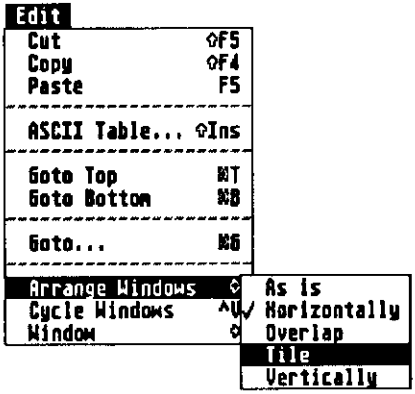
Hierarchical menu structures

A standard feature of the GEM desktop is the menu bar which appears at the top of the screen. This feature allows quick and easy access to key features of the application which is currently running. Traditionally, selection of extra features would be performed by a range of extra on screen dialog buttons or controls which may otherwise be available on the screen or in a window, but not any more...

....a new feature of AES 3.3 is that it is now possible to generate programs which can produce an attached sub-menu to the side of any of the menu elements. A sub-menu is activated by simply holding the mouse pointer on the element for a short period of time or by moving the mouse off the right hand side of that element, a secondary menu can be made to appear. Menu items which have a sub menu attached to them should indicate the fact by using an appropriate ASCII character on the far right hand side of the menu item thus:

An item with a sub-menu attachment

When on the screen, a sub-menu appears as a list of text items, similar in style to the original menu alongside which the sub-menu is placed thus:



A typical sub-menu

Movement of the mouse up and down the list will cause a black bar to follow it in order to show which item will be selected when the mouse is pressed. To close the sub-menu it is necessary to either select an item by clicking the mouse on an item or to move off the menu altogether and click some where outside, where upon it will close both the sub-menu and its parent (and leave the previous selection unaffected).

In principle it is possible to cascade sub-menus to a maximum depth of four. In practice however, it is not recommended that too many sub-menus are accessed from one another. This system is intended to increase the speed, ease of use and the overall software appeal. Programs which are poorly designed and use sub-menus to bad effect can produce a contrary effect by making the software less intuitive and slower to use. Applications which repeatedly force the user down a blind alley of menu lists cause great frustration in the user. We recommend that you don't use more than one or two levels of sub-menus attached to a single menu item.

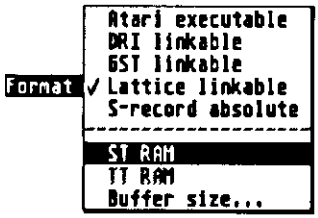
Pop-up menus

Once again, traditional GEM application design would dictate that when presenting the user with an array of selectable options, the decisions would be made by placing a dialog of some type on the screen. It is now possible to replace complex and cluttered user forms with a more basic range of choices using pop-up menus. This is because it is now possible to move all of the options for a function onto one neat menu list of their own. Frequent users of the famous radio button system will appreciate this feature immediately.

A typical pop up menu will have a piece of text on the screen describing a function. To the right of it there may be another piece of text surrounded by a box. This is the currently selected parameter for that function. For example:

A function with current status

By moving the mouse over this box and pressing the left mouse button, a sub-menu will appear:



A function with parameter selection

Notice how the currently selected item (which is ticked here) appears immediately to the right of the function name. It is a feature that the dialog display will be centred around the currently selected item. As with the sub-menu system, moving the mouse up and down the list will cause a highlight to follow it. Selection of an item is made by moving the mouse over the desired item (turning it black) and clicking the left mouse button on it. At this point the menu will disappear from the screen and the display will once again show the function description, but the contents of the box to its right may have changed, showing the new operative condition.

Scrolling in pop-ups and sub menus

When the user is presented with a sub-menu or pop-up menu, the number of items which appear in the list is determined by two factors. These are quite simply the number of items in the list (which may be only three or four items) or a limiting value which is passed as a parameter by the programmer to the call. If the number of items in the list exceeds the limiting factor, then the display will feature an arrow at the top and/or the bottom of the display to indicate that there are more items which can be displayed. The off screen items are revealed by clicking the mouse on the appropriate indicator, where upon the display within the menu will scroll up or down as appropriate, one item at a time. Scrolling will stop when the list of items to be displayed is exhausted or when the mouse is moved away from the associated arrow (presumably onto one of the options).

Colour & animated icons

The resource file system used by many applications has been updated by Atari. It is now possible to produce colour icons which change their appearance when they are selected (i.e. animated). This is most graphically demonstrated by the latest versions of the GEM desktop which feature colour icons in place of the old black and white ones. When an old icon was selected, it simply turned black, now the whole icon can be changed for another. For example an icon denoting a floppy disk may be used to represent drives A: or B:. By clicking on this icon, it is now possible to make the cover of the disk slide open denoting that it is in use.

In principle, the icons can be of almost any size and use any number of colours in their definition. In practice however small icons take less space (in both the RAM and screen area senses) and more than 16 colours in the definition are rarely worth while.

What's in AES 3.40...

3D controls on dialogs and windows

The standard appearance of applications can be changed by using new features of the AES. These pertain to the dialog boxes and standard GEM windows which are commonly used within programs. Within the new 3D system, GEM buttons and window sliders appear in a metallic grey colour and now feature a bevelled edge effect to make them appear stand proud of their background. This effect is further enhanced with a darker and lighter grey shade combination around the edges to provide a shadow. Moving the mouse onto a 3D button or window slider and clicking the mouse on it will cause the object to move slightly giving an impression of being pressed down. This feature provides the user with a more interactive form of response when using an application, as well as looking smarter than the older style of GEM program.

What's in AES 4.00...

The key feature of AES version 4.00 onwards is the modification and addition to the system to cater for multi-tasking applications. These include improvements to existing calls such as `appl_init` and the inclusion of totally new calls such as `appl_find`. To the end user, AES 4.00 does not provide any real visual indication of its presence within their machine (unlike AES 3.30 onwards) other than the fact that the multitasking is available for use.

The system calls

The AES functions all communicate with the OS via several arrays, the most useful of these to the user is the global array, named `_AESglobal` for Lattice C users, `global` for Devpac 3 users and a pointer to which is available via the GB function for BASIC users. The elements of this are:

`_AESglobal[0]`	AES version number in major minor form.
`_AESglobal[1]`	Number of concurrent applications the AES supports, or -1 if no limit (i.e. MultiTOS)
`_AESglobal[2]`	Application identifier for this application
`_AESglobal[3-4]`	User global, a longword global available for use by the user.

_AESglobal[5-6]	Pointer to base of resource file loaded as the result of a rsrc_load call.
_AESglobal[7-14]	Reserved.

Application library extensions

appl_find Find an application's identifier (4.00)

BASIC 2 FUNCTION appl_find%(BYVAL ap_fpname$)
FUNCTION appl_xfind%(BYVAL ap_fpname&)

Devpac 3 appl_find ap_fpname.L
Output Arguments
ap_id = int_out[0]

Lattice C int appl_find(const char *ap_fpname);

This call will find the ID of another application in the system. Normally `ap_fpname` will point to a NUL-terminated string containing the filename of the application for which the current application is searching. The string must be 8 characters long; if the filename has fewer than 8 characters, the programmer must fill out the rest of the string with blank spaces.

If running on AES ≥ 4.00 several additional features are available:

If the high word of `ap_fpname` is 0xFFFF, then the low word should contain the MiNT ID of that application. The `appl_find` call will convert the MiNT ID into the corresponding AES ID. If the high word of `ap_fpname` is 0xFFFE, then the low word should contain the AES ID of that application. The `appl_find` call will convert the AES ID into the corresponding MiNT ID.

If `ap_fpname` is a NULL pointer, then the `appl_find` will return the AES ID of the current process.

BASIC 2 users should note that two different calls exist, one accepts a string, the other a long word argument. The latter of these two calls is provided so that it is easier to access the new features of the call (see next paragraph) by masking in the relevant bit patterns into the argument. This prevents the requirement of lots of heavy string manipulation.

If the return is -1 then the AES could not find the requested application.

BASIC 2 FUNCTION appl_getinfo%(BYVAL ap_gtype%, VARPTR ap_gout1%, VARPTR ap_gout2%, VARPTR ap_gout3%, VARPTR ap_gout4%)

Devpac 3 appl_getinfo ap_gtype.W
Output Arguments
ap_greturn = int_out[0]
ap_gout1 = int_out[1]
ap_gout2 = int_out[2]
ap_gout3 = int_out[3]
ap_gout4 = int_out[4]

Lattice C int appl_getinfo(int ap_gtype, short *ap_gout1, short *ap_gout2, short *ap_gout3, short *ap_gout4);

This function allows an application access to general information about the AES. The value passed to the call, `ap_gtype`, tells the call what type of return is required. This call returns a value where a value of 0 represents an error and a value of 1 represents no error.

The `ap_gout...` parameters take on values based on `ap_gtype`: for `ap_gtype` is 0, get standard AES font information:

`ap_gout1`	font height in pixels
`ap_gout2`	font ID
`ap_gout3`	font type: 0 system font 1 Speedo font

If `ap_gtype` is 1, information about the small AES font is returned:

`ap_gout1`	font height in pixels
`ap_gout2`	font ID
`ap_gout3`	font type: 0 system font 1 Speedo font

`ap_gtype` is 2 returns the current resolution number and the number of colours supported by the object library:

`ap_gout1`	resolution number
`ap_gout2`	number of colours supported by AES object library
`ap_gout3`	colour icons: 0 Not supported 1 Supported
`ap_gout4`	new resource file format: 0 Not supported 1 Supported

`ap_gtype` is 3 returns the AES language setting:

`ap_gout1`	currently used language 0 English 1 German 2 French 3 reserved 4 Spanish 5 Italian 6 Swedish

This function *requires* AES ≥ 4.00.

appl_read — *Read from message pipe (4.00)*

BASIC 2 FUNCTION appl_read%(BYVAL ap_rid%, BYVAL ap_rlength%, VARPTR ap_rpbuff&)

Devpac 3 appl_read ap_rid.W, ap_rlength.W, ap_rpbuff.L
Output Arguments
ap_rreturn = int_out[0]

Lattice C int appl_read (int ap_rid, int ap_rlength, void *ap_rpbuff);

This function can be used to read `ap_length` bytes into the memory pointed to by `ap_rpbuff` from an application's message pipe. The application's identifier is supplied in the `ap_id` parameter; this is usually obtained from the result of the `appl_init` call.

Note that it is strongly recommended that you read 16 bytes at a time because of the way in which the AES works internally.

For AES ≥ 4.00 if `ap_id` is -1 then this function will perform a read only if there is data in the message pipe. Otherwise, it will return immediately.

appl_search Search existing AES processes (4.00)

BASIC 2 FUNCTION appl_search%(BYVAL ap_smode%, ap_sname$, VARPTR ap_stype%, VARPTR ap_sid%)

Devpac 3 appl_search ap_smode.W, ap_sname.L
Output Arguments
ap_sreturn = int_out[0]
ap_stype = int_out[1]
ap_sid = int_out[2]

Lattice C int appl_search(int ap_smode, const char *ap_sname, short *ap_stype, short *ap_sid);

This function, available in AES ≥ 4.00, searches all existing AES processes in the system for a process which matches defined criteria.

`ap_stype` indicates the required operation:

0	search first (all the processes)
1	search next (all the processes)
2	search system shell (only one)

`ap_sname` is a pointer to a buffer that will hold the name of the AES process the (which must at least 9 characters long). `ap_sid` is set to the AES ID of the process located, whilst `ap_stype` is set to the type of process found:

1	System process
2	Application
4	Accessory

The value returned is 0 to indicate no more files are available or non-zero otherwise.

This function *requires* AES ≥ 4.00.

Event library extensions

MultiTOS and the Falcon030 AES include a number of additional messages which may be returned from `evnt_mesag` or `evnt_multi` calls; these are documented below. These descriptions presume an 8 word buffer, `msg`, which has been set up by either `evnt_mesag` or `evnt_multi` as part of message being received; `msg[0]` will contain the message number of the message being delivered:

WM_UNTOPPED (30) **Current window untopped (3.30)**

This message is sent when the current top window is being untopped by another window. Note that by the time the message is received by the owner of the formerly topped window, that window is unlikely to still be on top. `msg[3]` contains the window handle of the window which has been removed from the topmost position.

WM_ONTOP (31) **New window on top (3.30)**

This message is sent when an application's window is placed on top, generally through no action of its own (i.e. another window closing). Note that by the time the message is received by the owner of the formerly topped window, that window is unlikely to still be on top. `msg[3]` contains the window handle of the window which has been removed from the topmost position.

AP_TERM (50) **Request application terminate (4.00)**

This message is sent when someone has requested that the shell manager close all active processes; for instance, when the user requests a resolution change. In response to this message the application should proceed with a normal termination sequence (such as close and delete all windows, remove the menu bar then exit).

If the process is unable to terminate it must inform the AES using `shel_write` mode 10.

In order for this message to be delivered to a process the application must have requested delivery of such messages via `shel_write` mode 9.

`msg[5]` contains a code indicating the reason that the shutdown was requested: AP_TERM (50) for a normal shut down, or AP_RESCHG (57).

AP_TFAIL (51) **Fail to terminate or close (4.00)**

This message should be sent by the receiver of an AP_TERM or AC_CLOSE message when it decides not to close or terminate.

The message send should have the form:

```
msg[0] = AP_TFAIL;
msg[1] = <your error code>;
```

which should be sent to the AES using `shel_write(10, ...)`.

SHUT_COMPLETED (60) System in shutdown state (4.00)

This message is sent to the process which initially requested the system be put into shutdown mode via `shel_write(4, ...)`. `msg[3]` indicates whether the shutdown was completed successfully: 1 if successful, 0 if it failed. If there is an error state `msg[4]` contains the AES ID of the process which could not terminate and `msg[5]` contains the error code which it returned.

If the shutdown was successful then on receipt of the system has been shutdown.

RESCH_COMPLETED (61) Resolution changed (4.00)

This message is sent to the process which initially requested the system be put into resolution change mode via `shel_write(5, ...)`. `msg[3]` indicates whether the resolution change was completed successfully: 1 if successful, 0 if it failed. If there is an error state `msg[4]` contains the AES ID of the process which could not terminate and `msg[5]` contains the error code which it returned.

If there is no error, then the caller must exit the system in order to complete the resolution change process.

AP_DRAGDROP (63) Desktop drag 'n' drop (4.00)

This message is sent by the desktop (or any other application) to another application in order to inform that application that the user has dragged an object to one of its windows, or that the user wishes it to open a new window. The message buffer received by the application has the form:

```
msg[0] = AP_DRAGDROP
msg[1] = sender's AES ID
msg[3] = window, or -1 for a request for a new window
msg[4] = mouse x
msg[5] = mouse y
msg[6] = keyboard state
msg[7] = pipe name
```

A thorough discussion of the drag and drop protocol is beyond the scope of this document.

SH_WDRAW (72) Request Desktop window redraw (4.00)

This message may be sent by applications to the Desktop to inform it that it should update its windows for a certain drive. `msg[3]` indicates which drive: 0 for drive A:, 1 for drive B:, etc. If `msg[3]` is -1, then the desktop will update all of its windows.

CH_EXIT (80) Child termination (4.00)

This message is sent back to a process which created a child via `shel_write` upon its termination. `msg[3]` contains the child process's AES ID, `msg[4]` gives the child's exit code.

graf_mouse — Change mouse form (4.00)

BASIC 2 SUB graf_mouse(BYVAL gr_monumber%,
BYVAL gr_mofaddr&)

Devpac 3 graf_mouse gr_monumber.W [, gr_mofaddr.L]
Output Arguments
gr_moreturn = int_out[0]

Lattice C int graf_mouse(int gr_monumber, const void *gr_mofaddr);

This call changes the appearance of the mouse on screen to that of one of a predefined set or to an application defined style. `gr_monumber` identifies the form to be used:

0	arrow
1	text cursor
2	busy bee
3	hand with pointing finger
4	flat hand, extended fingers
5	thin cross hair
6	thick cross hair
7	outline cross hair
255	mouse form stored in `gr_mofaddr`
256	hide mouse form
257	show mouse form

Up until AES 4.00, applications enjoyed free access to the `graf_mouse` function to change the mouse form. However, in the new multitasking environment, many applications may be present in the system at the same time, this may pose a problem. Once the mouse form has been changed to something other than the arrow, ownership is transferred to that application until it changes the mouse back to an arrow. So, as a courtesy to other applications, the current owner should always change the mouse back to arrow as soon as it finishes with its work.

However, in some circumstances, an application may want to change mouse form immediately without any delay. For example, the current foreground application changes the mouse to a busy bee and user clicks on the background to do a drag operation on a different application. It is very logical that the mouse should be changed to a flat hand for the dragging. In this case, the AES provides a way to force the current mouse to the next owner in order to deal with this type of situation; the following additional `gr_monumber` codes are available:

258	save current mouse form
259	restore to the last saved mouse form
260	restore to previous mouse form

In the event that the application *must* change the mouse form, it should set the highest bit (bit 15) of `gr_monumber` and OR in the desired mouse form number. After finishing the work, it should then call the `graf_mouse` with value 0 to set the mouse back to arrow.

Object library extensions

Colour Icons

AES 3.30 and above include substantial additions to deal with colour icons and which have the ability to take advantage of all resolutions and to perform limited animation when an icon is selected. Below is a description of the actual data structure; the structure itself follows:

```
typedef struct cicon_data {
 short num_planes;              /* num of planes */
 short *col_data;               /* ptr to bitmap */
 short *col_mask;               /* ptr to mask */
 short *sel_data;               /* ptr to sel icon */
 short *sel_mask;               /* ptr to sel mask */
 struct cicon_data *next_res;   /* ptr to nxt icon */
} CICON;

typedef struct cicon_blk {
 ICONBLK monoblk;               /* default mono icon */
 CICON *mainlist;               /* list of col icons */
} CICONBLK;
#define G_CICON 33              /* AES obj type */
```

The AES object library uses the `CICONBLK` structure to hold the data that defines colour icons. The object type `G_CICON` points with its `ob_spec` pointer to a `CICONBLK` structure.

CICONBLK is a colour icon block; a colour icon block contains a monochrome icon block and a list of colour icons. The list, is a linked list of colour icons that supports different resolutions. The monochrome icon block, monoblk, is the default icon displayed when mainlist does not contain an icon for the current resolution. Furthermore, the monochrome icon and all of the colour icons in mainlist share the dimensions, placement, and all textual information contained in monoblk.

CICON is the structure that contains the colour data. A CICON can contain pointers to two sets of icon data: one for the colour icon, col_data, and one for the colour icon in its selected state, sel_data. In both cases, the data is an array of words, and is in device-independent format. The number of planes of data is determined by num_planes (note that in mainlist, CICONs must have a unique num_planes.). Each CICON must have a valid pointer to data in col_data, but sel_data is optional. In other words, if sel_data is NULL, then when the icon is selected, the icon will be drawn darkened (i.e. dithered). Both col_data and sel_data pointers have masks: col_mask and sel_mask, respectively. Any other CICONs (with a different number of planes) are pointed to by next_res.

Three Dimensional Objects

Three dimensional objects are implemented in AES ≥ 3.40. Note that the 3D effects were first implemented in AES 3.30 using a different (and incompatible) method which is no longer supported.

The AES uses 2 new bits in the ob_flags field to indicate what kind of 3D shading effects (if any) should be used on the object. These are bits 9 and 10 of ob_flags. Versions of HiSoft's WERCS resource editor from version 1.25 onwards can be used to edit these bits directly. If both bits are clear (FL3DNONE) then no 3D shading effects are applied to the object.

In Lattice the defines would be as follows (note that these are present in AES.H):

```
#define    FL3DMASK    0x0600
#define    FL3DNONE    0x0000
#define    FL3DIND     0x0200
#define    FL3DBAK     0x0400
#define    FL3DACT     0x0600
```

If (ob_flags & FL3DMASK) is FL3DIND, then the object is an "indicator." Typically indicators are used in dialog boxes to indicate some sort of state; for example, whether an option is 'on' or 'off'. Radio buttons should always be indicators.

If (ob_flags & FL3DMASK) is FL3DACT, then the object is an "activator." Activators don't have a persistent state, but rather are usually controls of some sort. For example, the 'OK' and 'Cancel' buttons in dialog boxes should be activators.

If (ob_flags & FL3DMASK) is FL3DBAK, then the object is a "background" object. Background objects are usually not selectable, and do not typically display 3D effects other than inheriting the "3D background object" colour (see below); the only 3D effect applied to background objects is that "outlined" background objects appear to be raised above the objects behind them.

The colours (and effects, for indicators and activators) of 3D objects may be controlled by the objc_sysvar function. Any 3D object which is colour 0 (white) and has a hollow fill pattern will be drawn in the 3D default colour set for its object type, instead of in white. 3D objects which are not white or which have a non-hollow fill pattern will be drawn in the colour and pattern specified, as usual.

objc_sysvar Get/Set 3D colours and effects (3.40)

BASIC 2
```
FUNCTION objc_sysvar%(BYVAL ob_smode%,
BYVAL ob_swhich%, BYVAL ob_sival1%,
BYVAL ob_sival2%, VARPTR ob_soval1%,
VARPTR ob_soval2%)
```

Devpac 3
```
objc_sysvar ob_smode.W, ob_swhich.W, ob_sival1.W,
ob_sival2.W
Output Arguments
    ob_sreturn = int_out[0]
    ob_soval1 = int_out[1]
    ob_soval2 = int_out[2]
```

Lattice C
```
int objc_sysvar(int ob_smode, int ob_swhich, int ob_sival1,
int ob_sival2, short *ob_soval1, short *ob_soval2);
```

This call allows an application to set or inquire the colours and effects for 3D objects; applications should not change 3D colours or effects except at the explicit request of the user, because all such changes are global (i.e. they affect all processes).

ob_smode should be 0 to get the current attributes, or 1 to set them. ob_swhich controls which attributes are being set or inquired; this also affects the meaning of the other values, as follows:

LK3DIND 1	Get/set attributes for indicator objects. If ob_smode is 0, then ob_soval1 indicates whether the text of indicator objects does (1) or does not (0) move when the object is selected, and ob_soval2 indicates whether the object does (1) or does not (0) change colour when selected. If ob_smode is 1, then ob_sival1 controls whether indicator object text will (1) or will not (0) move when the object is selected, and ob_sival2 controls whether indicator objects will (1) or will not (0) change colour when selected. The default is ob_sival1 = 1 and ob_sival2 = 0.
LK3DACT 2	Get/set attributes for activator objects. The meanings of ob_soval1, ob_soval2, ob_sival1, and ob_sival2 are the same as for LK3DIND, except that they apply to activator objects rather than indicator objects. The defaults for activator objects is ob_sival1 = 0 and ob_sival1 = 1.
INDBUTCOL 3	Get/set default colour for indicator objects; this is the colour which hollow, white indicator objects (e.g. buttons) will be drawn in instead of white. If ob_smode is 0, then ob_soval1 is the current colour index of the default indicator object colour. If ob_smode is 1, then ob_sival1 is the new colour index for indicator objects.
ACTBUTCOL 4	Get/set default colour for activator objects. This call works in a similar way as the INDBUTCOL call, but applies to activators rather than indicators.
BACKGRCOL 5	Get/set default colour for 3D background objects. Same as INDBUTCOL, but applies only to 3D background objects.

AD3DVALUE 6	Get pixel adjustments for 3D indicators and activators. `ob_smode` must be 0. `ob_soval1` is set to the number of pixels by which 3D indicators and activators are expanded on each side horizontally (to accommodate 3D effects), and `ob_soval2` is the number of pixels by which they are expanded vertically. Remember that this adjustment is applied to each side of the object, so the objects width or height is increased by twice this amount. Background 3D objects never change in size.

This call will return 0 if the call failed or 1 if it was successful

This function *requires* AES ≥ 3.40.

Menu library extensions

menu_attach — Attach sub-menu (3.30)

BASIC 2 FUNCTION menu_attach%(BYVAL me_flag%, BYVAL me_tree&, BYVAL me_item%, BYVAL me_mdata&)

Devpac 3 menu_attach me_flag.W, me_tree.L, me_item.W, me_mdata.L
Output Arguments
me_return = int_out(0)

Lattice C int menu_attach(int me_flag, OBJECT *me_tree, int me_item, MENU *me_mdata);

This call allows an application to attach, change, remove or inquire about a submenu associated with a menu item. `me_flag` is the action to be performed by `menu_attach`:

0	Inquire about the submenu that is associated with the menu item. The data concerning the submenu is returned in `me_mdata`.
1	Attach or change a submenu associated with a menu item. `me_mdata` must be initialised by the application. The data must consist of the object tree of the submenu, the menu object, the starting menu item and the scroll field status. Attaching `NULL` will remove the submenu associated with the menu item.
2	Remove a submenu associated with a menu item. `me_mdata` should be set to `NULL`.

me_tree is the address of the object tree to which the sub-menu is to be attached. me_item is the object index of the menu item within the me_tree object which is to being attached to. me_tree is a pointer to a MENU structure:

```
typedef struct _menu {
 OBJECT *mn_tree;    /* object tree of menu */
 short mn_menu;      /* parent object of menu items */
 short mn_item;      /* starting menu item */
 short mn_scroll;    /* scroll field status of menu */
 short mn_keystate;  /* CTRL, ALT, SHIFT Key states */
}MENU;
```

Note that there can be a maximum of 64 associations per process. A menu item with an attached submenu uses the high-byte of its object type field; values 128 to 192 are used by the submenu menu system.

This call will return 0 if the call failed or 1 if it was successful

Beware that in all versions of the AES available at the time of writing any attempt to interrogate, change or remove an existing sub-menu (i.e. any operation other than initial attachment) will crash the machine.

This function *requires* AES ≥ 3.30.

menu_bar — Manage AES menu bar (4.00)

BASIC 2 SUB menu_bar(BYVAL me_btree&, BYVAL me_bshow%)

FUNCTION menu_xbar%(BYVAL me_btree&,
BYVAL me_bshow%)

Devpac 3 menu_bar me_btree.L, me_bshow.W
Output Arguments
me_breturn = int_out[0]

Lattice C int menu_bar(OBJECT *me_btree, int me_bshow);

This function informs the AES that it should use the object me_btree as its menu bar if the me_bshow parameter is 1. If me_bshow is 0 the menu bar is 'removed'; note that this does not actually erase the bar from the screen.

For AES ≥ 4.00 if me_bshow is -1, the menu_bar will become an inquiry call in which it will return the current menu owner's AES process ID. If the return value is -1, then there is no menu bar owner. BASIC owners should use the new menu_xbar function for this.

It is important to point out that the current menu bar can be swapped out at any time. If the application wants to update or redraw its menu bar, it is recommended to first check to see if it still own the menu bar and then proceed to its functions. However, the menu bar owner can still be changed after `menu_bar` call.

menu_istart Get/Set starting submenu item (3.30)

BASIC 2 FUNCTION menu_istart%(BYVAL me_flag%,
BYVAL me_tree&, BYVAL me_imenu%,
BYVAL me_item%)

Devpac 3 menu_istart me_flag.W, me_tree.L, me_imenu.W,
me_item.W
Output Arguments
me_return = int_out(0)

Lattice C int menu_istart(int me_flag, OBJECT *me_tree,
int me_imenu, int me_item);

This call allows an application to get or set or the starting item of a submenu. When the mouse is clicked onto a menu to which a submenu is attached, the submenu is shifted vertically so that the starting item is aligned with the mouse on the screen. `me_flag` gives the action to be performed:

0	Inquire which item is the starting item for the submenu
1	Set the starting item for the submenu to be `me_item`

`me_tree` is the address of the object tree to which the sub-menu is attached. `me_imenu` is the object index of the menu item within the `me_tree` object which is attached.

This call will return 0 if the call failed or 1 if it was successful

This function *requires* AES ≥ 3.30.

menu_popup — Display popup menu (3.30)

BASIC 2 FUNCTION menu_popup%(BYVAL me_menu&,
BYVAL me_xpos%, BYVAL me_ypos%,
BYVAL me_mdata&)

Devpac 3 menu_popup me_menu.L, me_xpos.W, me_ypos.W,
me_mdata.L
Output Arguments
me_return = int_out(0)

Lattice C int menu_popup(const MENU *me_menu, int me_xpos,
int me_ypos, MENU *me_mdata);

This call is used to display a popup menu and retrieve the user's response. `me_menu` is a pointer to a MENU structure:

```
typedef struct _menu {
 OBJECT *mn_tree;    /* object tree of menu */
 short mn_menu;      /* parent object of menu items */
 short mn_item;      /* starting menu item */
 short mn_scroll;    /* scroll field status of menu */
 short mn_keystate;  /* CTRL, ALT, SHIFT Key states */
}MENU;
```

If `mn_scroll` is not 0, then the menu will scroll if the number of menu items exceed the menu scroll height. The value is the object at which scrolling will begin; this will allow one to have a menu in which the scrollable region is only a part of the whole menu. The value must be a menu item in the menu.

`me_xpos` and `me_ypos` define the top-left edge of the starting menu item to be displayed. `me_mdata` is a pointer to a second MENU structure; if `menu_popup` returns a non-zero value, `me_mdata` contains information about the submenu that the user selected. This includes the object tree of the submenu, the menu object, the menu item selected and the scroll field status for this submenu.

The function returns 0 to indicate that the user did not click on an enabled menu item, or non-zero otherwise.

This function *requires* AES ≥ 3.30.

menu_register — Set AES program name (4.00)

BASIC 2 FUNCTION menu_register(BYVAL me_rapid%,
BYVAL me_rpstring$)

Devpac 3 menu_register me_rapid.W, me_rpstring.L
Output Arguments
me_rmenuid = int_out[0]

Lattice C int menu_register(int me_rapid, const char *me_rpstring);

This function is used to insert a menu entry for a desk accessory in the Desk menu. The text for the menu entry is passed as the `me_rpstring` parameter and the application identifier (`me_rapid`) is as returned from the `appl_init` call.

For AES ≥ 4.00 applications may call `menu_register` to change the name that appears in the menu bar for that application.

menu_settings — Set menu parameters (3.30)

BASIC 2 SUB menu_settings(BYVAL me_flag%,
BYVAL me_values&)

Devpac 3 menu_settings me_flag.W, me_values.L
Output Arguments
me_return = int_out[0]

Lattice C int menu_settings(int me_flag, MN_SET *me_values);

This call allows an application to set or inquire the sub-menu delay and scroll height values. `me_flag` is zero to inquire the current settings or 1 to get the settings into `me_values`. `me_values` is a structure of the form:

```
typedef struct _mn_set {
 long Display;      /* submenu display delay */
 long Drag;         /* submenu drag delay */
 long Delay;        /* single-click scroll delay */
 long Speed;        /* continuous scroll delay */
 short Height;      /* menu scroll height */
} MN_SET;
```

This function *requires* AES ≥ 3.30.

Resource library extensions

rsrc_rcfix — Fix pre-loaded resource file (4.00)

BASIC 2 SUB rsrc_rcfix(BYVAL rc_header&)

Devpac 3 rsrc_rcfix, rc_header.L
Output Arguments
rc_return = int_out[0]

Lattice C short rsrc_rcfix(void *rc_header);

`rsrc_rcfix` fixes up raw resource data that is already loaded into the memory by the application, pointed to by `rc_header`. It converts all the object locations and sizes into pixel co-ordinates. The resource must be the same as those generated by the resource construction set. If there is another resource already loaded into the system for the application, the application is required to do a `rsrc_free` to free up the memory before calling this function. The application still needs to perform an `rsrc_free` before termination.

This function *requires* AES ≥ 4.00.

Shell library extensions

shel_get — Read the AES's internal shell buffer (4.00)

BASIC 2 FUNCTION shel_get%(BYVAL sh_gbuff&,
BYVAL sh_glen%)

Devpac 3 shel_get sh_gbuff.L, sh_glen.W
Output Arguments
sh_greturn = int_out[0]

Lattice C int shel_get(char *sh_gbuff, int sh_glen);

This function reads the AES's internal shell buffer (the RAM version of the DESKTOP.INF/NEWDESK.INF file) into the buffer at the given address; `sh_glen` bytes will be read. The buffer should be at least 4192 bytes long to accommodate for TOS's later than AES version 1.40 (Rainbow TOS).

The function returns 0 if an error occurred or non-zero otherwise.

For AES ≥ 4.00 if `sh_glen` is -1 the value returned by this call is the amount of RAM required for the current AES shell buffer.

shel_put — *Write the AES's internal shell buffer (4.00)*

BASIC 2 SUB shel_put(BYVAL sh_pbuff&, BYVAL sh_plen%)

Devpac 3 shel_put sh_pbuff.L, sh_plen.W
Output Arguments
sh_preturn = int_out[0]

Lattice C int shel_put(const char *sh_pbuff, int sh_plen);

This function writes into the AES's internal shell buffer (the RAM version of the DESKTOP.INF/NEWDESK.INF file) from the buffer at the given address. `sh_glen` bytes will be written. The length must not be greater than 1024 bytes for AES versions prior to 1.40 (Rainbow TOS) or 4192 bytes for later TOS's. If you write a new buffer to the AES, you must place a single ^Z (26 decimal) to indicate the end of the buffer.

The function returns 0 if an error occurred or non-zero otherwise.

For AES ≥ 4.00 if `sh_glen` is larger that the current buffer size the AES will reallocate the memory for the buffer.

shel_write — *Run another application (4.00)*

BASIC 2 FUNCTION shel_write%(BYVAL sh_wdoex%, BYVAL sh_wisgr%, BYVAL sh_wiscr%, BYVAL sh_wpcmd&, BYVAL sh_wptail&)

Devpac 3 sh_wdoex.W, sh_wisgr.W, sh_wiscr.W, sh_wpcmd.L, sh_wptail.L, shel_write.W
Output Arguments
Parameters:
sh_wreturn = int_out(0)

Lattice C int shel_write(int sh_wdoex, int sh_wisgr, int sh_wiscr, const char *sh_wpcmd, const char *sh_wptail);

This function can be used to run another program when this application has finished. The `sh_wdoex` parameter should be 1 to run another program.

The `sh_wisgr` parameter specifies whether the program to be run is a .TOS (or .TTP) program (use 0 for this parameter) or a GEM (i.e. .PRG or .APP) program.

The sh_wpcmd parameter specifies the complete filename (including extension) of the program to be run. The sh_wptail parameter specifies the command tail to be used in GEMDOS Pexec format i.e. the first byte gives the length of the string.

The sh_wiscr parameter should be 1 to run the program when control returns to the Desktop.

For AES ≥ 4.00 shel_write has a huge number of extensions implemented as additional sh_wdoex values. For program launching the following sh_wdoex values are available (note that more values are described later):

0	Launch program. The GEM/TOS value will be determined by the AES.
1	Launch a TOS or GEM application.
2	Reserved
3	Launch an accessory.

If sh_wdoex is 0, the AES will determine the actual launching mode by looking at the file's extension. What file extensions are considered for launching is determined by the AES environment variables GEMEXT, TOSEXT, and ACCEXT.

The sh_wisgr parameter is used as above and indicates whether a GEM or TOS application is being launched; it is only valid when sh_wdoex is 1.

When a TOS program is launched under MultiTOS (via shel_write) the AES looks for the TOSRUN environment variable; this should be a full path of a TOS handler program to which the AES will pass the program name as the command tail.

sh_wiscr is used to set whether the AES constructs an ARGV style parameter list as part of the environment of the launched program. This should be 0 to disable ARGV generation, or 1 otherwise.

In addition the high byte of `sh_wdoex` is used to provide a 'extended' mode, i.e. a program or accessory may be launched in a customised way. In extended mode, `sh_wpcmd` is treated as a pointer to a set of long (32 bit) values. Each value after the first corresponds to one of the bits in `sh_wdoex`: if that bit is set then the corresponding value is used, otherwise it is ignored. The values and their associated bit numbers are as follows:

Offset	Bit	Function
`LONG[0]`		Pointer to the program name string (must be the first element)
`LONG[1]`	8	`Psetlimit` value.
`LONG[2]`	9	`Prenice` value.
`LONG[3]`	10	Default directory string pointer.
`LONG[4]`	11	Application defined environment string pointer.

The directory path (`LONG[3]`) should look something like:

```
C:\ or C:\FOLDER or C:\FOLDER1\FOLDER2 ...
```

However, if the pointer is zero, then the default directory will be the directory in which the program itself was found.

The value returned from this function when launching a program is the AES ID of the new process, or zero if an error occurred.

In addition to program launching `shel_write` provides several AES control mechanisms:

4	Set shutdown mode
5	Attempt resolution change
6	Reserved
7	Send a message to all processes
8	Alter AES environment
9	Inform the AES of an applications new message ability
10	Send the AES a message

If sh_wdoex is 4, the system is put into shutdown or normal mode depending on the sh_wisgr value; once the AES is in the shutdown mode, the shel_write launch capability (mode 0-3) is disabled. The following sh_wisgr values are used:

0	Abort the shutdown sequence; note that only the original caller of shutdown mode can abort the shutdown sequence.
1	Partial shutdown mode; the AES will check for all applications excluding the caller to make sure they all recognise AP_TERM message. If succeeded, AES will then send out AP_TERM to applications and AC_CLOSE to accessories. Note that the caller will receive none of the messages.
2	Complete shutdown mode; the AES will check for all applications and accessories excluding the caller to make sure they all recognise AP_TERM message. If successful, the AES will then send out AP_TERM to applications and AC_CLOSE to accessories. Accessories also receive AP_TERM after the AC_CLOSE message. Note that the caller will receive none of the messages.

In order for an application to receive AP_TERM messages it must notify the AES using shel_write mode 9.

sh_wdoex set to 5 requests that the AES change resolution, if the resolution change is accepted by the AES the system is put into shutdown mode upon which applications can either shut down and exit or deny the shutdown by sending an AP_TFAIL message to the AES.

If sh_wiscr is zero, then sh_wisgr is the physical device ID of the VDI workstation which should be opened.

If sh_wiscr is one, then sh_wisgr is the video mode word for use on Falcon030.

sh_wiscr value from 2 and up are reserved for future use.

sh_wdoex is 7 broadcasts a message to all processes except the AES, screen manager and the sender. In this mode, sh_wpcmd should point to a 16 byte message buffer. sh_wisgr and sh_wiscr are ignored.

sh_wdoex is 8 allows applications to manipulate the AES environment variables. sh_wisgr determines the operation to be performed: if sh_wisgr is 0 the function returns the environment buffer size in bytes.

For sh_wisgr is 1 the string pointed to by sh_wpcmd is added to the AES environment; this should have the form 'NEW=STRING\0'. If the string has the form 'NEW=\0' then the corresponding environment variable is removed.

For sh_wisgr is 2 the AES environment is copied into the buffer pointed to by sh_wpcmd. The output buffer size is specified by sh_wiscr. The return value indicates the number of bytes not copied.

sh_wdoex is 9 informs the AES of which additional messages the application can recognise via the sh_wisgr variable; the following bits are used:

0	AP_TERM
1...15	Reserved

sh_wdoex is 10 sends a message to the AES; sh_wpcmd is the 16 byte message buffer. Typically this may be used by an application to send an AP_TFAIL message to the AES after it finds it is unable to successfully shutdown upon an AES shutdown request.

Window library extensions

wind_get ***Get window attributes (3.30)***

BASIC 2 SUB wind_get(BYVAL wi_ghandle%, BYVAL wi_gfield%, VARPTR wi_gw1%, VARPTR wi_gw2%, VARPTR wi_gw3%, VARPTR wi_gw4%)

Devpac 3 wind_get wi_ghandle.W, wi_gfield.W
wind_get wi_ghandle.W, wi_gfield.W, wi_gw1.W, wi_gw2.W, wi_gw3.W, wi_gw4.W
Output Arguments
wi_gw1 = int_out[0]
wi_gw2 = int_out[1]
wi_gw3 = int_out[2]
wi_gw4 = int_out[3]

Lattice C int wind_get(int wi_ghandle, int wi_gfield, short *wi_gw1, short * wi_gw2, short *wi_gw3, short * wi_gw4);

This function returns information about a window given by wi_ghandle depending on the value of the parameter wi_gfield. The values returned, wi_gw1, wi_gw2, wi_gw3, wi_gw4 depend on wi_gfield; the additional wi_gfields are:

WF_TOP (10)	Returns handle in wi_gw1, owner's AES id in wi_gw2 and the handle of the window below it in wi_gw3 This function *requires* AES ≥ 3.30.

WF_NEWDESK (14)	Get the current system background window's object pointer. The value is returned in wi_gw1 and wi_gw2. This function *requires* AES ≥ 3.30.
WF_COLOR (18)	Get the window's element colour by handle. The value are returned in wi_gw2 and wi_gw3. `wi_gw1` is also used as an *input* parameter to this function and should contain the window element number required. This function *requires* AES ≥ 3.30.
WF_DCOLOR (19)	Get the default element colour. The values are returned in wi_gw2 and wi_gw3. `wi_gw1` is also used as an *input* parameter to this function and should contain the window element number required. This function *requires* AES ≥ 3.30.
WF_OWNER (20)	Get the window owner's AES id, window open status.: wi_gw1 — AES id of the owner wi_gw2 — 1 if the window is open, else 0 wi_gw3 — handle of the window above wi_gw4 — handle of the window below Note that if the window is closed, the `wi_gw3` and `wi_gw4` values will be meaningless. This function *requires* AES ≥ 3.30.
WF_BEVENT (24)	Get special window attributes ; see the `wind_get` function for definition of the bits returned in `wi_gw1`. This function *requires* AES ≥ 3.31.
WF_BOTTOM (25)	This function finds the current bottom window handle. This function *requires* AES ≥ 3.31.

BASIC 2 SUB wind_get(BYVAL wi_shandle%, BYVAL wi_sfield%, BYVAL wi_sw1%, BYVAL wi_sw2%, BYVAL wi_sw3%, BYVAL wi_sw4%)

Devpac 3 wind_set wi_shandle.W, wi_sfield.W, wi_sw1.W, wi_sw2.W, wi_sw3.W, wi_sw4.W

Lattice C int wind_get(int wi_shandle, int wi_sfield, int wi_sw1, int wi_sw2, int wi_sw3, int wi_sw4);

This function sets a window attribute given by `wi_sfield` for the window `wi_shandle`. The values `wi_sw1`, `wi_sw2`, `wi_sw3` and `wi_sw4` depend on `wi_gfield`.

`WF_BEVENT (24)`	This field sets attributes for the window; only bit 0 is defined at the time of writing which, if set, a button click in that window's work area will not cause a `WM_TOPPED` message to be sent to that window. Instead, the button click will satisfy `evnt_button` or the button-click option of an `evnt_multi` call. All other bits should be zero for future compatibility. This function *requires* AES ≥ 3.31.
`WF_BOTTOM (25)`	This function mode sets an already opened window to the bottom of the window stack (excluding the background window) and brings the next logical window to top. However, if the target window is the only open window in the system, this window will still remain on top and be active. This function *requires* AES ≥ 3.31.

wind_update ***Manipulate AES semaphores (4.00)***

BASIC 2 FUNCTION wind_update%(BYVAL wi_ubegend%)

Devpac 3 wind_update wi_ubegend.W

Lattice C int wind_update(int wi_ubegend);

This function is used to stop the user using menus, moving windows etc. whilst the application is outputting to the screen or when the application wants to do its own tracking of the mouse. These routines must called strictly in pairs; note that they do nest, so that so long as the calls match there are no problems.

For AES ≥ 4.00 a new check and set mode is defined for BEG_UPDATE and BEG_MCTRL, obtained by ORing 0x100 into `wi_ubegend`. The function returns 0 if the `wind_update` failed, or a non-zero positive integer on success.

Chapter 8
SpeedoGDOS

SpeedoGDOS provides VDI extensions to allow manipulation of high quality outline fonts, in addition to device independent output that has always been part of GDOS; this section documents the new calls designed to support the Speedo font scaling system.

Speedo data types

In order to represent fractional values, SpeedoGDOS uses a new data type: a signed 32-bit number with 1-bit sign and 31-bit magnitude; for example:

$00010000 1.0 pixels

$FFFF0000 -1.0 pixels

$00018000 1.5 pixels

Applications can do all calculations using simple long arithmetic, and completely avoid using a floating point library. To convert these units to integers (using C, but with exactly the same principle for Devpac and BASIC - note that fix31 is a data type of type `long`):

```
fix31 big_x;
int small_x;

small_x = big_x >> 16;
```

To round a value you can simply add 32778:

```
small_x = (big_x + 32768) >> 16;
```

v_bez — Output bézier

BASIC 2
```
SUB v_bez(BYVAL count%, BYVAL xyarr%(),
BYVAL bezarr&, BYVAL extent%(), VARPTR totpts%,
VARPTR totmoves%)
```

Devpac 3
```
v_bez count.W
Input Arguments
    xyarr = ptsin[0...count*2-1]
    bezarr = intin[0...count/2-1]
Output Arguments
    totpts = intout[0];
    totmoves = intout[1];
    extent[0] = ptsout[0];
    extent[1] = ptsout[1];
    extent[2] = ptsout[2];
    extent[3] = ptsout[3];
```

Lattice C
```
void v_bez(int handle, int count, const short *xyarr,
const char *bezarr, short extent[4], short *totpts,
short *totmoves);
```

`v_bez` is used to draw an unfilled bézier curve. `xyarr` points to a set of `(x, y)` co-ordinate pairs, whilst `bezarr` lists the attributes of each of the corresponding points. The following bits are used in `bezarr`:

Bit	Value	Meaning
0	0	Point begins a polyline section.
0	1	Point is the first point of a set of 4 bézier points in the sequence: first anchor point, first control point, second control point, second anchor point.
1	1	Point is a jump point; the current point is moved to this point without a joining line

The total number of points in the resulting polygon is returned in `totpts`, whilst the total number of moves in the resulting polygon is returned in `totmoves`. The bounding box of the polygon is returned in `extent`.

Machine code programmers should pay particular attention to the formation of the data held in the block of data pointed to by `bezarr`. The data which is stored here must be stored in an Intel byte swapped form. This means that before making the call, the data held in this block must have all of the bytes which make the data words swapped, so that bytes 0 and 1, 2 and 3 (and so forth) become bytes 1 and 0, 3 and 2 etc. BASIC and C programmers do not need to worry about this since the relevant library routines will handle all of this transparently.

v_bez_fill — Output filled bézier

BASIC 2 v_bez_fill(BYVAL count%, BYVAL xyarr%(), BYVAL bezarr&, extent%(), VARPTR totpts%, VARPTR totmoves%)

Devpac 3 v_bez_fill count.W
Input Arguments
xyarr = ptsin[0...count*2-1]
bezarr = intin[0...count/2-1]
Output Arguments
totpts = intout[0];
totmoves = intout[1];
extent[0] = ptsout[0];
extent[1] = ptsout[1];
extent[2] = ptsout[2];
extent[3] = ptsout[3];

Lattice C v_bez_fill(int handle, int count, const short *xyarr, const char *bezarr, short extent[4], short *totpts, short *totmoves);

`v_bez_fill` is used to draw a filled bézier curve. `xyarr` points to a set of (`x`, `y`) co-ordinate pairs, whilst `bezarr` lists the attributes of each of the corresponding points. The following bits are used in `bezarr`:

Bit	Value	Meaning
0	0	Point begins a polyline section.
0	1	Point is the first point of a set of 4 bézier points in the sequence: first anchor point, first control point, second control point, second anchor point.
1	1	Point is a jump point; the current point is moved to this point without a joining line

The total number of points in the resulting polygon is returned in `totpts`, whilst the total number of moves in the resulting polygon is returned in `totmoves`. The bounding box of the polygon is returned in `extent`.

Machine code programmers should pay particular attention to the formation of the data held in the block of data pointed to by `bezarr`. The data which is stored here must be stored in an Intel byte swapped form. This means that before making the call, the data held in this block must have all of the bytes which make the data words swapped, so that bytes 0 and 1, 2 and 3 (and so forth) become bytes 1 and 0, 3 and 2 etc. BASIC and C programmers do not need to worry about this since the relevant library routines will handle all of this transparently.

v_bez_off — Disable bézier capabilities

BASIC 2 SUB v_bez_off

Devpac 3 v_bez_off

Lattice C void v_bez_off(int handle);

This call disables the GDOS bézier capabilities. Any memory allocated by the GDOS for bézier-generated polygons is released at this time (see `v_set_app_buff` for memory allocation information).

Note that failure to disable bézier facilities prior to closing the (virtual) workstation may cause the VDI to crash.

v_bez_on — Enable bézier capabilities

BASIC 2 FUNCTION v_bez_on%

Devpac 3 v_bez_on
Output Arguments
retval = intout[0]

Lattice C int v_bez_on(int handle);

This call enables the GDOS bézier capabilities; note that while a handle is provided and the associated device driver is called, the GDOS bézier extension is enabled for all devices when this call is made.

The value returned from the call represents the maximum bézier depth, a measure of the smoothness of the curve. The value, which can range from 0 to 7, is an exponent of 2, giving the number of line segments that make up the curve. Thus, if `retval` is 0, the curve is actually a straight line (one line segment); if `retval` is 7, the curve is made of 128 line segments.

v_bez_qual — *Set bézier quality*

BASIC 2 SUB v_bez_qual(BYVAL percent%, VARPTR actual%)

Devpac 3 v_bez_qual percent.W
Output Arguments
actual = intout[0]

Lattice C void v_bez_qual(int handle, int percent, short *actual);

`v_bez_qual` sets the bézier speed/quality trade-off parameter as a percentage of the quality (hence 100% is best quality but slowest). The quality factor is passed in `percent` as a value from 0 to 100. The value selected by GDOS (approximated to the 8 levels available) is returned.

v_flushcache — *Flush outline font cache*

BASIC 2 FUNCTION v_flushcache%

Devpac 3 v_flushcache
Output Arguments
ret_val = intout[0]

Lattice C int v_flushcache(int handle);

This function flushes the contents of the outline font cache. Note that this only flushes the portion of the cache that contains bitmaps of outline font characters. The function zero normally, or -1 if an error has occurred.

BASIC 2
```
SUB v_ftext(BYVAL x%, BYVAL y%, VARPTR string$)

SUB v_ftext_offset(BYVAL x%, BYVAL y%,
VARPTR string$, VARPTR offset%())

SUB v_wc_ftext(BYVAL x%, BYVAL y%, BYVAL string&)

SUB v_wc_ftext_offset(BYVAL x%, BYVAL y%,
BYVAL string&, VARPTR offset%())

SUB v_wc_gtext(BYVAL x%, BYVAL y%, BYVAL string&)

SUB v_wc_justified(BYVAL x%, BYVAL y%,
BYVAL string&, BYVAL len%, BYVAL word%,
BYVAL chr%)
```

Devpac 3
```
v_ftext string.L
v_ftext_offset string.L
Input Arguments
    x = ptsin[0]
    y = ptsin[1]
    ptsin[2] = offset[0]
    ptsin[3] = offset[1]
    ...
    ptsin[n] = offset[n-2];
    ptsin[n+1] = offset[n-1];
```

Lattice C
```
void v_ftext(int handle, int x, int y, const char *string);

void v_ftext_offset(int handle, int x, int y,
const char *string, const short *offset);

void v_wc_ftext(int handle, int x, int y, const short *string);

void v_wc_gtext(int handle, int x, int y, const short *string);

void v_wc_justified(int handle, int x, int y,
const short *string, int len, int word, int chr);
```

This family of functions are used to display text on the output device (either explicitly referenced by `handle` for C or implicitly for Devpac/BASIC). The string to write is passed in `string` and it is displayed starting at position (`x`, `y`).

The v_...ftext_... family of functions work in exactly the same fashion as the more familiar v_gtext, but the text generated accounts for the fractional values of vqt_advance32; i.e. the text spacing is more accurate.

In addition, for applications that require more control over character placement, the v_ftext_offset and v_wc_ftext_offset functions take a custom set of offset vectors, one for each character in the string (including the last one for underlining calculations). Each vector consists of a pair of 16-bit values that are used in place of a character's advance vector when outputting text.

To support full 16 bit wide strings the additional v_wc_... bindings exist which expect a pointer to a string of 16-bit wide characters (i.e. each character occupies 16 bits rather than 8). For the v_...ftext_... the functions perform as above; for the v_wc_justified and v_wc_gtext functions the string parameter is modified as noted, but the other parameters are as in the older v_justified and v_gtext bindings.

If you are using the wide character routines you will probably want to request operation in the BICS mode from vst_charmap thus making all characters in the set available.

v_getbitmap_info
Get character bitmap information

BASIC 2 SUB v_getbitmap_info(BYVAL ch%, VARPTR advancex&, VARPTR advancey&, VARPTR xoffset&, VARPTR yoffset&, VARPTR width%, VARPTR height%, VARPTR bitmap&)

Devpac 3 v_getbitmap_info ch.W
Output Arguments
width = intout[0]
height = intout[1]
advancex = intout[2-3]
advancey = intout[4-5]
xoffset = intout[6-7]
yoffset = intout[8-9]
bitmap = intout[10-11]

Lattice C void v_getbitmap_info(int handle, int ch, fix31 *advancex, fix31 *advancey, fix31 *xoffset, fix31 *yoffset, short *width, short *height, short **bitmap);

This call provides information to allow the caller to know the exact size and placement of a given character. This information includes the character's x and y advance vectors, the x and y offsets, and the bitmap dimensions of the character. The advance vector represents the amount to add to the current point to place the following character. The x and y offsets, when added to the current point, give the caller the location of the upper left hand corner of the bitmap. The width and height of the bitmap are returned as 16 bit integers. All other values are returned in fix31 representation.

v_getoutline — Get character outline

BASIC 2 SUB v_getoutline(BYVAL ch%, VARPTR xyarray%, BYVAL bezarray&, BYVAL maxverts%, VARPTR numverts%)

Devpac 3 v_getoutline ch.W, xyarray.L, bezarray.L, maxverts.W
Output Arguments
numverts = intout[0]

Lattice C void v_getoutline(int handle, int ch, short *xyarray, char *bezarray, int maxverts, short *numverts);

This function generates an outline of the character specified by `ch`, in the current character set (see `vst_charmap`), and places the bézier representation of that character into the buffers provided by the caller. This bézier information can readily be sent to the GDOS bézier output call since `xyarray` and the `bezarray` are exactly the information required by the GDOS bézier primitive. `maxverts` is the total number of vertices that the user's buffers will handle. `numverts` represents the number of vertices contained in the given bézier.

v_loadcache — Load outline font cache

BASIC 2 FUNCTION v_loadcache%(VARPTR filename$, BYVAL mode%)

Devpac 3 v_loadcache filename.L, mode.W
Output Arguments
ret_val = intout[0]

Lattice C int v_loadcache(int handle, const char *filename, int mode);

This function loads in the contents of a outline font cache from disk. The function takes a filename and a mode as parameters. The filename specifies what file to open in the current directory. The `mode` specifies whether or not to append or create a new cache. If the `mode` is 0, the cache from disk will be appended to the current cache; if it is 1, then the cache will be flushed, and a new cache will be loaded. The function returns zero normally, or -1 if an error has occurred.

v_savecache — Save outline font cache to disk

BASIC 2 FUNCTION v_savecache%(VARPTR filename$)

Devpac 3 v_savecache filename.L
Output Arguments
ret_val = intout[0]

Lattice C int v_savecache(int handle, const char *filename);

This function saves the contents of the outline font cache to disk. The function takes a filename as its parameter, and the cache is saved under that filename. The file is created in the current directory. The function returns zero normally, or -1 if an error has occurred.

v_set_app_buff — Reserve bézier workspace

BASIC 2 SUB v_set_app_buff(VARPTR address&, BYVAL nparagraphs%)

Devpac 3 v_set_app_buff address.L, nparagraphs.W

Lattice C void v_set_app_buff(void *address, int nparagraphs);

This call makes the nominated memory block available for use by the GDOS bézier extensions. When the application makes bézier calls, the buffer set aside by this call holds the polygon generated from the bézier anchor and direction points; if this call is not made, a default 8K buffer is allocated by GDOS.

`address` is a pointer to the memory block, and `nparagraphs` is the number of *paragraphs* available in the block (a paragraph is 16 bytes of memory).

For the C binding note that *no workstation handle* is passed.

Note that since no workstation handle is passed to this call the buffer is implicitly available for *all* applications in the system; if you are running under MultiTOS then you must nominate the allocation type of this memory block as supervisor access.

vqt_advance, vqt_advance32
Inquire outline font text advance vector

BASIC 2 SUB vqt_advance(BYVAL ch%, VARPTR advx%, VARPTR advy%, VARPTR xrem%, VARPTR yrem%)

SUB vqt_advance32(BYVAL ch%, VARPTR advx&, VARPTR advy&)

Devpac 3 vqt_advance ch.W
Output Arguments
advx = ptsout[0]
advy = ptsout[1]
xrem = ptsout[2]
yrem = ptsout[3]
vqt_advance32 ch.W
(Additional output)
advx = ptsout[4-5]
advy = ptsout[6-7]

Lattice C void vqt_advance(int handle, int ch, short *advx, short *advy, short *xrem, short *yrem);

void vqt_advance32(int handle, int ch, fix31 *advx, fix31 *advy);

This function returns the `x` and `y` offsets which are needed to place the next character of a string in the proper position. This call is necessary when laying down text at rotations other than 0, 90, and 270 degrees.

Remainder values are returned in two different ways, either in `xrem` and `yrem` (for `vqt_advance`), or as `fix31` values for (`vqt_advance32`). Note that the `fix31` has greater accuracy and the programmer is urged to use these values instead.

vqt_cachesize — Get outline font cache size

BASIC 2 SUB vqt_cachesize(BYVAL which_cache%, VARPTR size&)

Devpac 3 vqt_cachesize which_cache.W
Output Arguments
size = intout[0-1]

Lattice C void vqt_cachesize(int handle, int which_cache, long *size);

`vqt_cachesize` obtains the size of one of the Speedo font caches. The size of the cache required is dictated by the `which_cache` parameter; this is 0 to find the size of the largest space in the bitmap cache, or 1 to find the size of the largest space in the data structure cache. The size of the selected cache is returned in `size`.

vqt_devinfo — Inquire device status information

BASIC 2 SUB vqt_devinfo(BYVAL devnum%, VARPTR devexists%, VARPTR devstr$)

Devpac 3 vqt_devinfo devnum.W
Output Arguments
devexists = ptsout[0];
devstr = intout;

Lattice C void vqt_devinfo(int handle, int devnum, short *devexists, char *devstr);

`vqt_devinfo` is used to ascertain whether a particular driver ID has been installed and what driver is associated with it. `handle` contains a workstation handle for the device. On return `devexists` is non-zero if the device exists, whilst `devstr` contains the ASCII name for the device.

Devpac developers please note: The string returned in `devstr` takes the form of the ASCII characters which go to make up the string, one character in each element of the intout array. That is to say that each character is passed back in a word format with the most significant byte of each word padded out with $00. Before use, the programmer will probably have to unpack the string into a separate storage area or buffer.

vqt_fontheader
Inquire Speedo font header information

BASIC 2 vqt_fontheader(BYVAL buffer&, VARPTR pathname$)

Devpac 3 vqt_fontheader buffer.W
Output Arguments
pathname[0] = intout[0];
pathname[n] = intout[n];

Lattice C vqt_fontheader(int handle, void *buffer, char *pathname);

This function copies the current font's Speedo font header into a buffer and returns the full path name for the corresponding TDF file. Note that the buffer must contain at least 420 bytes. For further information on the font header, please refer to the Speedo Font Header Appendix.

Devpac developers: please read the specific information at the start of this section regarding the format of returned strings.

vqt_f_extent Inquire outline font text extent

BASIC 2 SUB vqt_f_extent(VARPTR string$, VARPTR extent%())

Devpac 3 vqt_f_extent string.L
Output Arguments
extent[0] = ptsout[0]
...
extent[7] = ptsout[7]

Lattice C void vqt_f_extent(int handle, const char *string,
char extent[8]);

This function returns the screen area needed to display a string of graphics text using the current text attributes, taking *into account* any fractional information from `vqt_advance32`. This gives how much screen area will be used if `v_ftext` is used to display that string.

The diagram below shows how the points that mark the boundary of the string are numbered:

The extent information is returned in `extent`, which should be large enough to hold 8 words, will be returned as follows:

`extent[0]`	x co-ordinate of point 1.
`extent[1]`	y co-ordinate of point 1.
`extent[2]`	x co-ordinate of point 2.
`extent[3]`	y co-ordinate of point 2.
`extent[4]`	x co-ordinate of point 3.
`extent[5]`	y co-ordinate of point 3.
`extent[6]`	x co-ordinate of point 4.
`extent[7]`	y co-ordinate of point 4.

vqt_get_table — *Get character mapping tables*

BASIC 2 SUB vqt_get_table(VARPTR map&)

Devpac 3 vqt_get_table
Output Arguments
map = intout[0-1]

Lattice C void vqt_get_table(int handle, short **map);

This call returns the address of a series of contiguous tables used internally by SpeedoGDOS. The tables are used to map the Atari character set to the equivalent Bitstream character indexes. Depending on the font file, a Speedo font's indexes have six different formats: the Bitstream International Character Set, the Bitstream International Symbol Set, the Bitstream Dingbats Set, the PostScript text set, the PostScript symbol set, and the PostScript Dingbats set. There are a total of seven tables which map the Atari character set to Bitstream character indexes, one master mapping, and one table for each of the aforementioned character sets (in the order presented). Applications can find out which character set corresponds to the current font by checking the Speedo font file header.

Each individual table contains 224 word-sized entries with the first entry being the translation for character 32, the second for character 33....etc. Therefore, with the address of the table, applications can change the mappings so that any Bitstream character index may be substituted.

vqt_f_name — *Inquire face name and index*

BASIC 2 FUNCTION vqt_name%(BYVAL element_num%, VARPTR name$)

Devpac 3 vqt_name element_num.W
Output Arguments
index = intout[0]
name[0] = intout[1]
...
name[31] = intout[32]
name[32] = intout[33]
name[33] = fsmflag

Lattice C int vqt_f_name(int handle, int element_num, char name[32], short *isfsm);

This function returns the name of a font and its font index. The function that changes the current font, `vst_font`, requires a font index which should be obtained using `vqt_name/vqt_f_name`.

The font numbers that are passed in the `element_num` parameter start at 1 and are followed by 2, 3, etc. until the number of loaded fonts. The number of loaded fonts is returned by the `vst_load_fonts` call. Font number 1 is the system font.

In addition this call returns an additional parameter (when running under Speedo GDOS) which indicates whether the selected font is an outline font or an old-style bitmap font. The manner in which this parameter is accessed differs for all three languages.

For Devpac `intout[33]` set to 0 will signify a bitmap font, and 1 will indicate an outline font. Devpac developers should also read the specific information at the start of this section regarding the format of returned strings.

BASIC users should note that this function is part of the *normal* GEMVDI library and that `mid$(name$,33,1)` will give the outline flag.

For the C binding, `vqt_f_name` (cf. `vqt_name` for Devpac/BASIC) the final parameter, `isfsm`, is set to 1 by the binding to indicate that the selected font is an outline font. If the font is a bitmap font then 0 is returned. Note that Lattice C preserves the old binding `vqt_name` for backward compatibility.

vqt_pairkern Inquire pair kerning information

BASIC 2 SUB vqt_pairkern(BYVAL ch1%, BYVAL ch2%, VARPTR x&, VARPTR y&)

Devpac 3 vqt_pairkern ch1.W, ch2.W
Output Arguments
x = ptsout[0-1]
y = ptsout[2-3]

Lattice C void vqt_pairkern(int handle, int ch1, int ch2, fix31 *x, fix31 *y);

This function allows the application to inquire the adjustment vector for pair kerning. The `vqt_pairkern` function returns the vector (x-direction and y-direction) that indicates the spacing adjustment made between the character pair specified in `fix31` pixel units.

vqt_trackkern Inquire track kerning information

BASIC 2 SUB vqt_trackkern(VARPTR x&, VARPTR y&)

Devpac 3 vqt_trackkern
Output Arguments
x = ptsout[0-1]
y = ptsout[2-3]

Lattice C void vqt_trackkern(int handle, fix31 *x, fix31 *y);

This function allows the application to inquire the adjustment vector for track kerning. The `vqt_trackkern` function returns the vector (x-direction and y-direction) that indicates the spacing adjustment made between characters specified in `fix31` pixel units.

vst_arbpt, vst_arbpt32
Set character cell height by arbitrary points

BASIC 2 FUNCTION vst_arbpt%(BYVAL point%, VARPTR chwd%, VARPTR chht%, VARPTR cellwd%, VARPTR cellht%)

FUNCTION vst_arbpt32&(BYVAL point&, VARPTR chwd%, VARPTR chht%, VARPTR cellwd%, VARPTR cellht%)

Devpac 3 vst_arbpt point.W (integer point sizes)
vst_arbpt int_pnt.W, frac_pnt.W (fractional point sizes)
Output Arguments
set_point = intout[0][1]
chwd = ptsout[0]
chht = ptsout[1]
cellwd = ptsout[2]
cellht = ptsout[3]

Lattice C int vst_arbpt(int handle, int point, short *chwd, short *chht, short *cellwd, short *cellht);

fix31 vst_arbpt32(int handle, fix31 point, short *chwd, short *chht, short *cellwd, short *cellht);

This function selects an arbitrary point size for an Speedo font. This differs from the `vst_point` call which will only allow the sizes mentioned in the EXTEND.SYS file to be selected; note that this call will only work with outline fonts.

Two bindings are provided for C and BASIC; `vst_arbpt` which sets an exact integer point size, and `vst_arbpt32` which uses the Speedo `fix31` format to permit any fractional point size.

The value returned by the function indicates the point size which was actually selected, `chwd` and `chht` indicate the character size selected in raster co-ordinate units. `cellwd` and `cellht` indicate the cell size in raster co-ordinate units.

In the case of Devpac, note that the fractional size is expected in two separate words, the first being the integer part and the second being the fraction; if only one word is passed the macro will arrange for only the integer part to be used.

vst_charmap — Set character mapping mode

BASIC 2 SUB vst_charmap(BYVAL mode%)

Devpac 3 vst_charmap mode.W

Lattice C void vst_charmap(int handle, int mode);

This function allows the application to switch from using the Atari character set to the Bitstream International Character Set (BICS). The `vst_charmap` function will set a flag so that all subsequent calls to text calls will use words that are equivalent to Bitstream character indexes instead of Atari ASCII bytes. The `mode` flag requires the value of 0 for Bitstream mode and 1 for ASCII mode.

vst_error — Set SpeedoGDOS error mode

BASIC 2 SUB vst_error(BYVAL mode%, VARPTR errorcode%)

Devpac 3 vst_error mode.W, errorcode.L

Lattice C void vst_error(int handle, int mode, short *errorcode);

`vst_error` configures the way in which Speedo errors are reported. The default, `mode` is 1, places Speedo error messages on the screen. If `vst_error` is called with `mode` set to 0 then future errors are placed in the `errorcode` variable. The following error codes are used:

Value	Meaning
0	No error
1	Character not found in font
8	Error reading file
9	Error opening file
10	Bad file format
11	Out of memory/cache full
-1	Miscellaneous error

A SpeedoGDOS error may be generated by any of the following calls:

```
v_ftext, v_ftext_offset, v_wc_ftext, v_wc_ftext_offset,
v_gtext, v_justified, v_wc_gtext, v_wc_justified, v_opnvwk,
v_opnwk, vqt_advance, vqt_advance32, vqt_extent,
vqt_f_extent, vqt_f_name, vqt_fontinfo, vqt_name, vqt_width,
vst_arbpt, vst_font, vst_height, vst_load_fonts, vst_point,
vst_setsize, vst_unload_fonts
```

vst_kern — Set kerning mode

BASIC 2 SUB vst_kern(BYVAL tmode%, BYVAL pmode%, VARPTR tracks%, VARPTR pairs%)

Devpac 3 vst_kern tmode.W, pmode.W
Output Arguments
tracks = intout[0]
pairs = intout[1]

Lattice C void vst_kern(int handle, int tmode, int pmode, short *tracks, short *pairs);

This function allows the application to set the different kerning modes. Track kerning can have 0-3 tracks: 0 is no kerning, 1 is normal, 2 is tight, and 3 is very tight. The `vst_kern` function passes in a track kern value and it returns the track to which the current font is set. Pair kerning is set to be on or off.

`tracks` indicates the track kern mode set, whilst `pairs` gives the number of kerning pairs in the font.

vst_scratch — Set scratch buffer allocation mode

BASIC 2 SUB vst_scratch(BYVAL mode%)

Devpac 3 vst_scratch mode.W

Lattice C void vst_scratch(int handle, int mode);

This function sets the method of memory allocation for the scratch buffer. The scratch buffer is memory used to create text with special effects, and its size is determined by the maximum dimensions of a font. Since Speedo fonts can be scaled to any size, the scratch buffer size will not have limits.

Furthermore, many Speedo fonts don't need special effects (or a scratch buffer), because some of the fonts will actually be defined with certain effects already applied. By default, the allocation mode is 0, which takes Speedo fonts into account when calculating the scratch buffer size. If the mode is 1, the size will not be affected by Speedo fonts, and will take only bitmap fonts into account. In this case, special effects should not be used for Speedo fonts. If set to 2, no scratch buffer will be allocated, and special effects should not be used at all.

Also note that some VDI screen drivers (such as replacement video card drivers) may not support on-screen effects with Speedo fonts; for this reason, and the ones noted above, it is strongly recommended that you do not use algorithmic effects on Speedo fonts.

vst_setsize, vst_setsize32
Set character cell width by arbitrary points

BASIC 2 FUNCTION vst_setsize%(BYVAL point%,
VARPTR chwd%, VARPTR chht%, VARPTR cellwd%,
VARPTR cellht%)

FUNCTION vst_setsize32&(BYVAL point&,
VARPTR chwd%, VARPTR chht%, VARPTR cellwd%,
VARPTR cellht%)

Devpac 3 vst_setsize point.W (integer point sizes) or
vst_setsize int_pnt.W, frac_pnt.W (fractional point sizes)
Output Arguments
set_width = intout[0-1]
chwd = ptsout[0]
chht = ptsout[1]
cellwd = ptsout[2]
cellht = ptsout[3]

Lattice C int vst_setsize(int handle, int point, short *chwd,
short *chht, short *cellwd, short *cellht);

fix31 vst_setsize32(int handle, fix31 point, short *chwd,
short *chht, short *cellwd, short *cellht);

`vst_setsize` sets the graphic text character width in points. This allows an arbitrary set size to be used for the character width. Note that the next call to `vst_point`, `vst_arbpt` or `vst_height` will override any set size set by this call. This call will only work with SpeedoGDOS outline fonts. The set size may be specified in either 16-bit integer format (`vst_setsize`) or 32-bit fix31 format (`vst_setsize32`).

The value returned by the function indicates the set size which was actually selected, `chwd` and `chht` indicate the character size selected in raster co-ordinate units. `cellwd` and `cellht` indicate the cell size in raster co-ordinate units.

BASIC 2 FUNCTION vst_skew%(BYVAL skew%)

Devpac 3 vst_skew skew.W
Output Arguments
set_skew = intout[0]

Lattice C int vst_skew(int handle, int skew);

`vst_skew` sets the skew used when generating characters, note that this is independent of the skewing generated using `vst_effects`. The `skew` value (between -900 and 900) represents the number of 10^{ths} of a degree by which the characters are to be skewed. Negative values produce skews to the left, positive values skews to the right. Note that characters will degenerate badly when their skew approaches 90 degrees. This call only works with outline fonts.

The function returns the skew value actually set.

Appendix A
The Atari Style Guide

Since the announcement of the new multi-tasking environment for the Atari range of computers, Atari have published some useful software design guide lines for programmers. It is highly recommended that these comments should be read, understood and strictly adhered to for all future software and where possible, they should be implemented retrospectively into current available software. Programmers who do not heed the recommendations of the primary hardware manufacturers do so at their own peril since good software should be 'Future Proof', designing software to the following specifications will go part of the way towards that goal.

Finally, it is perhaps fair to say that some of the following documentation will appear to be a little pointless or just a matter of fine detail, especially when a program is being developed within a non multi-tasking environment. Please be advised that some of the guide lines only become apparent when working in a multi-application, multi-window environment.

Application Elements

User-friendly GEM applications should provide the user with a consistent, predictable means of interacting with the computer. The most popular applications to-date have always been those that the user feels at home with, because of general familiarity with other applications that they have previously used. User interface design is a critical consideration during product development and should be well thought out before actually sitting down and laying out and coding the interface.

The basic elements of a GEM application are the menu bar, the application's window (or windows), dialog boxes, alert boxes, and if the application warrants them, toolbox windows. GEM applications may optionally install their own desktop background, which is swapped out by the AES to reflect the active application.

The Menu Bar

Applications should normally consist of a MENU BAR, which will generally have the titles from left to right, "Prgname", "File", "Edit", and then the additional application-specific main menu titles. "Prgname" should be replaced with the application name so that users can quickly identify which application's menu bar they are looking at.

The File menu

For user convenience, the standard entries under "File" should start with "New", "Open...", followed by other load-oriented operations, then in the next section of the menu, "Close", "Save", "Save as...", and the other application-specific save-oriented functions. The next section down should be used for other file operations such as "Import..." and "Export...". This should be followed by the menu items for printing, usually "Page Setup...", then "Print...". The last item under "File" should always be "Quit".

Note: A menu item must be followed by an ellipsis to indicate that additional action or input will be required by the user to carry out the requested task. For instance, "Save" indicates that the file will be saved directly, using the current name, whereas "Save as..." will require the additional input of a filename.

The Edit menu

The "Edit" menu should start with "Undo", then in the next section, "Cut", "Copy", "Paste", and "Delete". The rest of the "Edit" menu is usually application-specific, but the next menu item, if used should be "Select all".

Other menus

If applicable, the fourth main menu title should be "Options", where menu items such as "Document defaults...", or "Preferences..." should appear.

Note: Menu titles and items should never be displayed in all uppercase letters. Menu titles should have one space before and after each title. There should be two spaces to the left of menu items.

Keyboard equivalents

Menu items

The standard keyboard equivalents that should be used system-wide for no other purpose other than those listed are:

[Control-N]	New
[Control-O]	Open
[Control-W]	Close
[Control-S]	Save
[Control-P]	Print
[Control-Q]	Quit
[Control-X]	Cut
[Control-C]	Copy
[Control-V]	Paste
[Control-A]	Select all
[Control-F]	Find
[Control-H]	Replace
[Control-G]	Find next
[Delete]	Delete
[Undo]	Undo
[Help]	Invoke help

Note: The [Alternate] key is used as a character modifier on non-U.S. keyboards to access the necessary extended characters in applicable countries, and should not be used for keyboard equivalents in most cases.

Cursor movement inside windows

The system-wide standard for keyboard cursor manipulation is as follows:

[Control-Left/Right Arrow]	Move cursor to beginning of word to the left/right
[Control-Backspace]	Delete from cursor position to start of next word to the left
[Control-Delete]	Delete from cursor position to start of next word to the right

[ClrHome]	Move cursor to beginning of document
[Shift-ClrHome]	Move cursor to end of document
[Shift-Delete]	Delete line

Windows

The primary stage for user interaction with the application is the window. Most of the user input, whether typing, drawing, or editing, is performed in the confines of windows. All of an application's output should be constrained to the application's own windows only. See the VDI and AES manuals for further information regarding window work areas and clipping rectangles.

Document windows should have, at a minimum, a mover/title bar so that even if the window is not resizable, the user can move the window off to the side of the desktop to have access to other items. The other window elements are the Info bar, Closer, Sizer, Full box, Sliders, and Arrows. The general use of these is apparent in the GEM Desktop. It should be noted that GEM sliders are always proportional so that the user has constant feedback as to the percentage of the document that is being viewed.

Operating system calls allow every element of windows to be set to any colour and fill pattern. The user generally selects these attributes using the Window Colours CPX in the Control Panel and they should not be altered by an application. In video modes with greater than 16 colours, other than True Colour, the first 16 colour entries should be reserved for use by the system for drawing elements for which the user has set preferences.

Dialog boxes

Dialog boxes are used for modal input. That is, input that the user must provide before any further processing may be done. They are generally used for parameter setting and other selections that require the undivided attention of the user. They should never be used for on-going informational or status output, as it would interfere with the normal real-time user interaction with the system.

Alerts

Alerts should be used to call the user's attention to conditions that develop that require immediate user knowledge. The simplest and most common would be an alert notifying the user that he is about to exit an application without having saved the open document. Alerts should also be used to notify the user that a time-consuming or unalterable function is about to be performed.

Alerts usually have two or three buttons that allow the user to make some sort of decision based on the information provided. Alerts with only one button are very frustrating to the user, as it implies a lack of control over what is about to happen. The general rule for alerts is to have the "OK" button to the left of the "Cancel" button. "Cancel" should always be capitalised, and "OK" is uppercase.

Note: In general, text within buttons should use capitalised words and should not all be in uppercase.

Toolbox windows

Toolbox Windows are a special class of window that are used for providing the user with non-modal control or information. The most common use would be for drawing tool selection in a paint program, or colour selection. The tools are usually shown as logical groups of icons that the user can easily associate with their functions. Another use of this type of window is continual status output, such as the progress of a file download or recalculation time.

Other general notes

Applications should make no assumptions about the type of system the user will have. They should be able to deal with any screen size and colour resolution. To achieve this the programmer should use the operating system calls to determine the screen dimensions and system capabilities to provide the user with the greatest flexibility possible.

Appendix B
Object File Formats

This section describes several of the object file formats ("linker formats") which are in common use under Atari's TOS.

Lattice

This section describes the Lattice linkable file format. A Lattice object file is a tag based file format, thus allowing particular fields to be present or absent from the file as required by the application. This also has the major advantage of (almost) limitless expandability.

Each tag is formed by a longword giving the type of the tag, followed by zero or more longwords giving additional information about the tag. Note that all items within a Lattice format file are longword sized unless noted otherwise.

A string in a Lattice format file is represented by a longword prefix giving the length *in longwords*, followed by the characters forming the string; as such the string may or may not be NUL terminated.

Module directives

The module directives are used to start and terminate object files with a file. A simple object file will have only one HUNK_UNIT, whereas an uncompressed library may have many.

HUNK_UNIT `0x3e7 string`

A *HUNK_UNIT* is used to introduce the start of a module within the relocatable file. It is followed by a string giving the name of the module. If there is more than one section within a module then HUNK_UNIT terminates the previous module.

Section directives

The section directives are used to start and end sections within a module. Each section (HUNK_CODE, HUNK_DATA and HUNK_BSS) may have a HUNK_NAME associated with it, followed by one or more relocation and/or debugging directives, terminated by a HUNK_END.

HUNK_NAME — 0x3e8 string

HUNK_NAME is used to associate a name with the next HUNK_CODE, HUNK_DATA or HUNK_BSS directive. It is followed by a string giving the name of the section. Note that section names are optional, although if present are used by the linker to coalesce sections of the same name. Unnamed sections are *always* coalesced.

HUNK_CODE — 0x3e9 long
HUNK_DATA — 0x3ea long

HUNK_CODE and *HUNK_DATA* define a block of code, and a block of data respectively. They have a long giving the number of longwords of code/data which follow the tag.

HUNK_BSS — 0x3eb long

HUNK_BSS defines a BSS block, containing `long` *longwords* of data. Note, however, that none of the data is actually present, and is implicitly zero.

HUNK_END — 0x3f2

A *HUNK_END* is used to mark the end of the current *section*. Note however that the smallest legal module is HUNK_UNIT, HUNK_END.

HUNK_CHIP — 0x40000000
HUNK_FAST — 0x80000000

HUNK_CHIP and *HUNK_FAST* are not section directives but section modifiers. They are bits which may be set in the HUNK_CODE, HUNK_DATA and HUNK_BSS section types to indicate that the respective section should be exclusively loaded into system (chip) or alternative (fast) memory respectively. Note that neither of these bits is supported by Lattice format linkers at the time of writing.

Relocation/symbol directives

The relocation directives are used to specify fixups which are to be performed in the current section. These may either be anonymous (i.e. not relative to any variable) in the case of the HUNK_RELOC8, HUNK_RELOC16, HUNK_RELOC32, HUNK_DRELOC8, HUNK_DRELOC16 and HUNK_DRELOC32 directives, or symbol relative in the case of the HUNK_EXT directive. Note that the HUNK_EXT directive is also used to generate exports of symbols.

HUNK_RELOC8	*0x3ee {long section offsets}*
HUNK_RELOC16	*0x3ed {long section offsets}*
HUNK_RELOC32	*0x3ec {long section offsets}*

HUNK_RELOC8, *HUNK_RELOC16* and *HUNK_RELOC32* directives specify 8, 16 and 32 bit relocations, respectively, which are to be performed in the current section. Each of these directives are followed by a relocation block consisting of a count of the number of relocations to be performed, which section the relocations are relative to and finally the relocation offsets. The list is terminated by a block indicating a zero count.

Each relocation blocks consist of a `long` giving the number of items to be fixed up relative to the `section` number. Note that the sections within the module are numbered starting from zero. Following `section` are a list of `long` longword offsets which are to be fixed up.

For HUNK_RELOC32 the relocation is performed by adding the base of the nominated section to the longwords located at the offsets. Note that there is no way to perform a 32 bit PC-relative fixup.

For HUNK_RELOC16 and HUNK_RELOC8 a PC-relative fixup is performed by adding the start of the section, minus the offset of the relocation point.

HUNK_DRELOC8	*0x3f9*
HUNK_DRELOC16	*0x3f8*
HUNK_DRELOC32	*0x3f7*

HUNK_DRELOC8, *HUNK_DRELOC16* and *HUNK_DRELOC32* directives specify 8, 16 and 32 bit near data section relocations, respectively, which are to be performed in the current section. Each of these directives are followed by a relocation block in the same format as for HUNK_RELOC32.

For *all* of these directives the relocation is performed by adding the base of the specified section (i.e. the number of bytes from the base of the __MERGED section to the base of the specified section).

Note the lack of symmetry between the HUNK_RELOC16, HUNK_RELOC8 and HUNK_DRELOC16, HUNK_DRELOC8 directives respectively.

HUNK_EXT 0x3ef { ... }

The *HUNK_EXT* directive is used to introduce a block specifying both definitions for symbols (exports) and references to declared symbols (imports).

Each symbol block within a HUNK_EXT block consists of an initial longword giving the length of the symbol which follows. The top 8 bits of the longword are ignored for the length purpose and instead hold a type. Immediately following this longword there are length longwords of name, giving the name of the symbol to which this symbol data unit refers.

EXT_ABS 2
EXT_DEF 1

The *EXT_ABS* and *EXT_DEF* types are used to enter absolute and relocatable definitions, respectively, into the linkers symbol table. Following the symbol name is a longword value. For an EXT_ABS this is the absolute value of the symbol. For an EXT_DEF it is an offset from the base of the current section to the symbol location.

EXT_REF8 132
EXT_REF16 131
EXT_REF32 129

The types *EXT_REF8*, *EXT_REF16* and *EXT_REF32* perform 8, 16 and 32 bit relocations, respectively, in the current section. Each of these is followed by a count of the number of references and a list of the offsets to these references.

EXT_REF32 references have the value of the symbol added to the longwords located at the offsets.

For EXT_REF16 and EXT_REF8 a PC-relative fixup is performed by adding the value of the symbol, minus the offset of the relocation point.

EXT_DREF8 *135*
EXT_DREF16 *134*
EXT_DREF32 *133*

The types *EXT_DREF8*, *EXT_DREF16* and *EXT_DREF32* perform 8, 16 and 32 bit near data section relocations, respectively, which are to be performed in the current section. Each of these is followed by a count of the number of references and a list of the offsets to these references.

For *all* of these types the relocation is performed by adding the offset of the symbol within the __MERGED.

Note the lack of symmetry between the EXT_REF32 and EXT_DREF32.

EXT_COMMON *130*

The *EXT_COMMON* type is used to enter a reference to a common block. Following the symbol there is a longword length for the common block. Note that this gives the length of the common block *in longwords*.

A list of references to the common symbol for fixup then follow in the same manner as used by EXT_REF32. Note that common references are *always* 32 bit.

Debugging directives

The debugging directives supply information which is not directly processed by the linker, but instead passed on to the final executable program for use by a symbolic or source level debugger.

HUNK_SYMBOL *0x3f0*

HUNK_SYMBOL is used to provide symbol information in the executable for use by a symbolic debugger.

Each symbol block within a HUNK_SYMBOL block consists of an initial longword giving the length of the symbol which follows. The top 8 bits of this longword are ignored for the length purpose and are always zero. Immediately following this longword there are length longwords of name, giving the name of the symbol to be placed in the executable symbol table. The longword offset from the start of the section immediately follows the name.

There may be as many of these symbol blocks as required, terminated by a longword 0 (indicating a zero length string).

HUNK_DEBUG *0x3f1*

HUNK_DEBUG is used to provide source level debugging information about the current section. It consists of the HUNK_DEBUG directive followed by a longword count of the number of longwords in the remainder of the HUNK_DEBUG section. The first longword in the debug block is filled in by the linker, by adding the offset of the start of this section within the entire coalesced section. The second longword is a type value, indicating the type of the debugging information contained in this block.

Note that there will typically be as many HUNK_DEBUG blocks within a section as there are source files which make up the module.

'HEAD' *0x48454144*

The *HEAD* chunk is generated by the linker as the very first HUNK_DEBUG directive in an executable file (see below for the location of HUNK_DEBUG information in an executable).

The first longword is the value 'DBGV' (0x44424756), followed by the HEAD chunk version number. This section describes version '01' (0x3031).

The word following the version number gives the linker option flags. Only two of these bits are committed at present: bit 0 indicates that the executable is an overlaid program (not supported by current linkers), bit 1 indicates that the file was linked in a case-insensitive manner.

The next two longwords give the number of symbols found in HUNK_DATA (or HUNK_BSS) sections and HUNK_CODE sections respectively.

The next longword indicates how many HUNK_DEBUG directives were encountered during linking (and hence how many follow the HEAD chunk), to improve the performance of locating these HUNK_DEBUG directives the linker writes an index of these immediately after the count of them. This consists of a longword, the high byte of which gives the number of the section which the HUNK_DEBUG refers to and the remainder of which gives an offset within the executable file to the start of the relevant HUNK_DEBUG directive. Note that this places a theoretical limit of 16Mb on the size of an executable plus debugging information.

'HBPR' *0x48425052*

The *HBPR* chunk is generated by HiSoft BASIC for use by the profiler. The information contained within this chunk is confidential and proprietary.

'HCLN' *0x48434c4e*

The *HCLN* chunk gives information about the line numbers of a file from which the module was generated. The first longword gives the length of a string giving the name of the file from which the module was generated, followed by length longwords of the name. Note that up to this point the format is identical to the LINE chunk.

The next longword gives the number of line number and offset pairs which are to follow. The remainder of the chunk then consists of signed line number and offset *delta* pairs, giving the changes in line numbers and code offsets, so that line numbers and code offsets may be matched. The encoding of the deltas is identical to that of the 680x0 BSR in its short, word and long forms. If a non zero byte (*note a byte*) is read then this is gives the delta, otherwise a word is read which, if non-zero, gives the delta, finally a longword may be read to give the delta. Note that the only way of generating a zero delta is via a byte, a word and finally a longword of zero (i.e. these should be avoided where possible).

'LINE' *0x4c494e45*

The *LINE* chunk gives information about the line numbers of a file from which the module was generated. The first longword gives the length of a string giving the name of the file from which the module was generated, followed by length longwords of the name.

The remainder of the chunk consists of line number and offset pairs, giving which offsets within the local code section correspond to which line numbers.

Note that there is no explicit terminator, instead the section ends when the length longword in the HUNK_DEBUG directive is used up.

'SRC ' *0x53524320*

The *SRC* chunk is used by Lattice C for source level debugging information. The first 8 longwords in the SRC chunk give the module name to which this HUNK_DEBUG refers (note that this is only the stem of the filename, *not* the entire file name). The next three longwords give (in bytes): the size of the code generated for this module, the size of the line number/offset pair table and the size of the source level debugging information table.

Next in the chunk a LINE format chunk is generated, comprising a longword giving the length of a string giving the name of the file from which the module was generated, followed by length longwords of the name. A line number/offset pair table then follows, of length given by value retrieved earlier from the chunk.

The next part of the chunk contains the source level debugging information table, of a size given earlier in the chunk. The contents of this part of the chunk are confidential and proprietary.

The final 5 longwords in the hunk are used to hold offsets to the near data, far data, near BSS, far BSS and chip data (not used under TOS). The values of these fields are modified by the linker to give the correct offsets after coalescing.

Library Format

The Lattice object format supports two library formats. The simple, uncompressed version is simply the concatenation of ordinary object modules (in an identical manner to that done in the GST format). There is a second, compressed, format recognised by CLink which allows a massive linker performance improvement.

HUNK_LIB 0x3fa

An *HUNK_LIB* directive is the first part of a compressed library. It is followed by a longword giving a count of the total number of longwords in the remainder of the block.

The data contained within a HUNK_LIB is simply the concatenation of all the modules which form the library, however all HUNK_UNIT and HUNK_NAME directives are removed completely whilst EXT_DEFs and EXT_ABSs are removed from HUNK_EXTs. Note that EXT_COMMONs (the other "definition" type) are not removed, but never-the-less appear within the HUNK_INDEX block.

Because of a restriction in the HUNK_INDEX directive, no section (HUNK_CODE, HUNK_DATA, or HUNK_BSS) in a HUNK_LIB may begin beyond a longword offset of 65535 (byte offset 262140). If this is required then a library may be enlarged by concatenating two or more compressed format libraries in the same manner as an uncompressed library.

Within the library file, the next directive, following a HUNK_LIB *must* be a HUNK_INDEX.

HUNK_INDEX 0x3fb

An *HUNK_INDEX* directive gives the module index for the preceding HUNK_LIB. It is followed by a longword giving a count of the total number of longwords in the remainder of the block.

The first item in the HUNK_INDEX is a string table, prefixed by an unsigned short word giving the number of bytes in the string table. The entries in the string table are module names (HUNK_UNIT), section names (HUNK_NAME) and external names (HUNK_EXT). Within the string table the strings are NUL-terminated (C-style) strings, concatenated end-to-end. Note that the first string in the table *must* be the NULL string. The string table is always padded so that it is an even number of words, although each string within the table is not padded.

Following the string table are one or more module entries. These give details of the modules contained within the HUNK_LIB. A module entry consists of an unsigned short offset to the HUNK_UNIT name within the string block, an unsigned short longword offset to the start of the module within the preceding HUNK_LIB, and an unsigned short count of the number of sections in the module (HUNK_CODE, HUNK_DATA and HUNK_BSS).

For each of the sections in the module, there is a section entry consisting of an unsigned short offset to the name of the section within the string block (note that 0 will indicate the NULL string), an unsigned short count of the number of longwords in the section within the HUNK_LIB and a short type of the section (HUNK_CODE, HUNK_DATA or HUNK_BSS).

The next unsigned short value indicates the number of references to external symbols (EXT_REF16, EXT_REF32, EXT_DREF16, EXT_DREF32 and EXT_COMMON), followed by a list of unsigned short offsets to the strings referenced within the string block. Note that a 16 bit reference is indicated by pointing to the zero *preceding* the string within the block, whereas a 32 bit reference points at the string itself (an EXT_COMMON is treated as an EXT_REF32). Note that EXT_REF8 and EXT_DREF8 are *not* supported.

The next unsigned short value indicates the number of external definitions (EXT_DEF, EXT_ABS and EXT_COMMON) within the section. Note that EXT_DEF and EXT_ABS definitions are deleted from the HUNK_EXT, whereas EXT_COMMONs are not (and indeed appear both as references *and* definitions). Following the count of references are a series of 6 bytes records.

The first unsigned short value gives an offset into the string block for the name of the symbol, followed by two unsigned shorts encoding the value and type of the symbol.

The encoding used for the value and type of the symbol is rather convoluted, but permits 25 bits of resolution for values and allows all current HUNK_EXT types.

The first unsigned short gives the bottom 16 bits of the value of the symbol. The bottom byte of the next unsigned short *excluding* bit 6 encodes the type byte (EXT_DEF, EXT_ABS or EXT_COMMON). The top 8 bits of this short then encode bits 16-23 of the value. Finally bit 6 is used to denote the state of the top 8 bits of the value, 0 if they are 0x00, or 1 to indicate 0xff. Thus the type and value of a definition may be decoded in the following manner:

```
unsigned short *buffer;
long value;
int type;

value = buffer[0] | ((buffer[1] & 0x0000ff00L) << 8);
if (buffer[1] & (1 << 6))
   value = value | 0xff000000;
type = buffer[1] & ~(1 << 6)
```

Note that there is no explicit terminator, instead the section ends when the length longword in the HUNK_INDEX directive is used up.

GST

This section describes the GST linkable file format. A GST object file is based around a byte stream which is copied by the linker to the final output file. Interspersed with the byte stream are directives, introduced using the code $FB which allow relocation, symbol definition, section and module definition. If a $FB code is required in the file then one is included using the sequence $FB $FB.

Within this file symbols and section names are referred to using an word-sized *id* number. These numbers are +ve and non-zero for symbols, whilst -ve values refer to SECTION and COMMON names. The value 0 is reserved for absolute section references.

A *string* in GST terminology is a sequence of bytes comprising the string, prefixed by the length of the string *excluding* the length byte (a.k.a. a Pascal string).

Source directives

Source directives are used to specify the start and end of modules within a single file. A GST format object module will have only one such pair of SOURCE and END directives; a library may have many appearing sequentially in the file. Note that a SOURCE directive must be the very first item in a GST format file.

COMMENT

FB 02 string

The *COMMENT* directive is used to include a comment within the relocatable file; it has no effect on the executable generated. Many linkers use this field type in conjunction with a librarian to improve performance of library scanning for GST format files.

DEFINE

FB 10 id string

The *DEFINE* directive is used to introduce id numbers for use by the other directives. It defines that the string given is associated with the id.

Remember that +ve (non-zero) id values are used for symbols, whilst -ve values refer to SECTION and COMMON names. The id value 0 is reserved for absolute section references.

END

FB 13

The *END* directive marks the end of the current module. If there are further modules in the file then these are introduced using a SOURCE directive.

SOURCE

FB 01 string

The *SOURCE* directive is used to indicate which source file the object file was derived from. and as such should only appear at the start of a module.

Section directives

The section directives allow the specification of various sections within a single module, e.g. a text, data and BSS section.

COMMON

FB 12 id

The *COMMON* directive is similar to the SECTION directive (see below) but instead switches to a section which rather than concatenating from module to module (as with a SECTION) overlay each other. The size of the resulting section is then the size of the largest one encountered.

Note that most existing linker implementations permit only non-initialised COMMON sections, i.e. the only valid output directive is OFFSET.

OFFSET

FB 05 long

The *OFFSET* directive moves the current location pointer within the section. The parameter gives the *absolute* offset within the section for the location pointer. Note that this may cause the section to be expanded as a result.

ORG

FB 03 long

The *ORG* directive indicates that the bytes following the directive are to start at the absolute address given in the long parameter. This applies until the next ORG, SECTION or COMMON directive.

SECTION

FB 04 id

The *section* directive switches the current section to that specified by id. The section should already have been defined via a *DEFINE* directive. this applies until the next *ORG*, *SECTION* or *COMMON* directive.

Symbol directives

The symbol directives are used to specify symbol definitions and fixups which should be performed for the relocatable file.

XDEF

FB 06 string long id

The *XDEF* directive marks symbol as an export. The value of the symbol is given by long, and is relative to the section defined by id.

XREF *FB 07* long trunc-rule {op id} *FB*

The *XREF* is the most complex directive. It is used to include fixup information in the final file. The final value output is constructed from the sum of the `long` parameter and the `op id` terms.

The `trunc-rule` parameter is a byte which defines the characteristics of the item finally written to the file. Six bits of it are used as follows:

Bit	Meaning
0	The result is byte sized
1	The result is word sized
2	The result is long sized
	Note that only one of these bits may be set.
3	The result is signed
4	The result is unsigned
	Unlike the size bits neither or both if these may be set in order to decide the range checking information.
5	The reference is PC relative, i.e. the location counter prior to the output of the result is subtracted from the running total.
6	The result should be relocated at runtime. This is only supported in TOS for longs.

The `op id` terms are a list of symbols which should be added/subtracted from the value. `op` is the character + for addition or - for subtraction. The following `id` (previously defined via an `id` directive) is then add/subtracted from the running total. The list is terminated by a single $FB byte.

Library Format

A GST format library is a relocatable object file as described above, but rather than containing only one SOURCE and END directive it will contain several. A library is therefore simply the concatenation of several object files.

DRI

This section describes the DRI linkable file format. A DRI object file consists of two images of the text and data segments (hence their voluminous size), the first giving the code, and the second the relocation information. The format is extremely similar to the GEMDOS executable format, with the sole difference occurring in the format of the relocation information. It should be noted that the format is incomplete since byte fixups are not possible.

Relocatable Format

A DRI relocatable file starts with a file header in the following format:

```
struct oheader {
        short magic;            0x601a
        long tsize;             length of text segment
        long dsize;             length of data segment
        long bsize;             length of BSS segment
        long ssize;             length of symbol table
        char reserved[10];      must be zero
};
```

Immediately after the header an image of the text section occurs, with length `tsize` and then the data section of length `dsize`. The symbol table then follows. This is an stream of symbols in the following format:

```
struct symbol {
        char name[8];           8 character name of symbol
        unsigned short type;    type of the symbol
        long value;             value assigned to symbol
};
```

Within these structures, `name` gives the 8 character name of the symbol. If the original name required more than 8 characters it will be truncated to 8 characters. `value` gives a value which is associated with the symbol. The `type` word is a used gives a bitmap of the types associated with the symbol as follows:

Bit	Meaning
15	defined, i.e. the value field gives the value the value of the to be associated with the symbol. Note that any of the other bits may be set to indicate a definition relative to that type.

14	equated label
13	global label
12	register equate, i.e. a symbol acting as a register synonym. The `value` field is used to indicate which register the symbol refers to; 0-7 indicate D0-D7, 8-15 indicate A0-A7 and 16-23 indicate FP0-FP7.
11	external symbol, i.e. a symbol imported from another module. Note that this means the value field would be unused. Because of this the format uses this symbol type to indicate a *common* symbol when the value field is *non-zero*. `value` then gives the length of the common block to be created.
10	value relative to start of DATA
9	value relative to start of TEXT
8	value relative to start of BSS

The relocation tables follow the symbol table. These are the same size as the section to which they refer and consist of words giving the relocation which is to be performed on the data. The top 13 bits of the relocation word give a symbol index number (note that this gives an 8192 symbol limit per-module), whilst the bottom 3 bits give the relocation type:

Type	Meaning
0	Absolute reference, no fixup required. Symbol ignored.
1	DATA segment relative, add address of local data segment. Symbol ignored.
2	TEXT segment relative, add address of local text segment. Symbol ignored.
3	BSS segment relative, add address of local BSS segment. Symbol ignored.
4	Absolute address of undefined symbol, add value of the symbol.
5	References the upper word of a long word to be relocated. Symbol ignored. Next word determines type of relocation and symbol.
6	PC relative, add symbol-PC.
7	References word which is never to be relocated (e.g. an instruction). Symbol ignored.

Within this scheme it is possible to generate all 16 and 32 bit relocation types, although most linkers will not permit a short absolute fixup (relocation type 4 without a preceding 6). This type of relocation is used by Lattice C to indicate an near data section reference.

Absolute Format

A DRI absolute file is identical to the relocatable format, with the exception of the header. For the absolute format it has the following structure:

```
struct abshdr {
        short magic;            0x601b
        long tsize;             length of text section
        long dsize;             length of data section
        long bsize;             length of BSS section
        long ssize;             length of symbol table
        long reserved;          zero
        long textbase;          absolute base of text section
        short relocflag;        zero if relocation present
        long database;          absolute base of data section
        long bssbase;           absolute base of BSS section
};
```

Following the header the format is identical to the relocatable format.

Note that GEMDOS *cannot* execute absolute format programs.

Executable Format

A GEMDOS executable file starts with a file header in the following format (note the similarity to the DRI relocatable module format):

```
struct prgheader {
        short magic;            0x601a
        long tsize;             length of text segment
        long dsize;             length of data segment
        long bsize;             length of BSS segment
        long ssize;             length of symbol table
        long reserved;          zero
        long flags;             program load flags
        short relocflag;        zero if relocation present
};
```

The bottom 28 bits of the `flags` longword are used to control the loading characteristics of the program. The bits are assigned as follows.

Bit	Meaning when set
0	Only clear the BSS of the program cf. the entire TPA.
1	Load the program into alternative RAM.
2	Allow `Malloc()` to take memory from alternative RAM.
4-5	Memory protection mode (MiNT): $00 Private (no reads or writes) $10 Global (any reads or writes) $20 Supervisor (supervisor mode reads and writes) $30 Shared (any reads, no writes)
11	Shared text segment (MiNT)

The top 4 bits of the flags longword are used in conjunction with the alternative RAM load bit. If this bit is set then TOS checks the amount of alternative memory against the value in the top 4 bits of `flags*128+128`, if more alternative RAM is available than this minimum TPA size then the program is loaded into alternative RAM, otherwise it is loaded into system RAM.

Note that prior to TOS 1.04, setting the `relocflag` word causes `Pexec()` to leave the file open after loading it; this behaviour usually results in two or more files with the *same name* appearing.

Immediately after the header an image of the text section occurs, with length `tsize` and then the data section of length `dsize`. The symbol table then follows, in almost the same format as the relocatable symbol format.

Note that within the symbol table all symbols are considered relative to the text section *regardless of the type field*.

As extensions to the standard format some utilities allow various extensions to the symbol types present in the executable symbol table:

Value	Meaning
0x??48	HiSoft extended symbol. This symbol type indicates that the next symbol record in the file is not a symbol, but instead an additional 14 characters to be added to the 8 characters from the name in the symbol record. The normal type information is in the top 8 bits.

0x0280	file symbol. A file symbol appears at the start of each object module in the symbol table, the `name` of which is the name of the module, and the `value` of which is the start of the text segment of that module. Immediately following the file symbol are all the symbols defined within the module.
0x02c0	library symbol. A library symbol is used to mark the start and end of libraries in the executable. At the start of the modules drawn from the library a library symbol occurs with the `name` giving the name of the library. After the end of the last symbol drawn from the archive another archive symbol is written with a blank name.

The relocation information differs completely from the DRI relocatable format. The relocation starts with a long word giving the first longword to be relocated. The byte stream that follows gives the offset to the next longword to be fixed. The values used are:

Value	Meaning
0	end of relocation information.
1	add 254 to the current location counter and then decode the next fixup byte
2..255	add value to location counter and fixup longword at that location.

Beyond the end of the relocation information there may be additional debugging information for use by symbol and source level debuggers. There is no standard representation for this information, although for HiSoft products this information (if present) is in the format of Lattice `HUNK_DEBUG` hunks, aligned to a word boundary.

Library Format

A DRI format library is formed by concatenating several DRI format object modules together with a header added at the front of each module. The initial word in the file is the magic number `0xff65` to indicate that the file is a library file. The header pre-pended to each module is as follows:

```
struct arheader {
        char a_fname[14];       module's file name
        long a_modtim;          last modified time of module
        char a_userid;          user ID (not used)
        char a_gid;             group ID (not used)
        short a_fimode;         file mode of module
```

```
        long a_fsize;           size of module in bytes
        short reserved;         zero
};
```

Within this structure, `a_fname` gives the name of the module, with any directory information removed. `a_modtim` gives the UNIX® format time stamp of the module (i.e. seconds since 00:00:00 GMT, January 1, 1970). `a_fimode` gives the UNIX® file mode bits; under TOS only one of these bits is used; bit 7 is set if write permission is available for the file, cleared otherwise. `a_fsize` gives the size of the module which follows in bytes. Note that this size *includes* the header pre-pended to the DRI format module, but *excludes* the archive header size.

Appendix C
The Cookie Jar

The Cookie Jar is a convention, introduced in STE TOS, whereby the system (and third party suppliers) can indicate the capabilities of an Atari TOS machine.

Each cookie has a 4 character name and a long integer value; cookies beginning with _ are reserved for Atari's system cookies. The long word at address $5A0 points to list of longword pairs. The first longword in a pair is a 4 character name; the second word is a value corresponding to that name. The list is terminated by a 0 long word as the name; note that cookies beginning with _ are reserved for Atari's system cookies.

Although the cookie jar was introduced with STE TOS it can be retro-fitted to earlier STs so don't assume that if there is a Cookie Jar then you are running on at least and STE.

For Falcon TOS and MiNT a number of additional cookies are available; below is a list of all Atari supplied cookies at the time of writing.

_AKP	Atari keyboard preference; this indicates the preferred keyboard layout and language preferences:		
	USA	0	USA
	FRG	1	Germany
	FRA	2	France
	UK	3	United Kingdom
	SPA	4	Spain
	ITA	5	Italy
	SWE	6	Sweden
	SWF	7	Switzerland (French)
	SWG	8	Switzerland (German)
	TUR	9	Turkey
	FIN	10	Finland
	NOR	11	Norway
	DEN	12	Denmark
	SAU	13	Saudi Arabia
	HOL	14	Holland
	CZE	15	Czechoslovakia
	HUN	16	Hungary
	This cookie is used by any machine using AES ≥ 3.30.		

_CPU	the bottom 2 digits of the main processor number (e.g. $0 for 68000, $1E for 68030)
_FDC	This gives an indication of the highest density floppy unit installed in the machine. The high byte of its value indicates the highest density floppy present: 0 360Kb/720Kb (double-density) 1 1.44Mb (high-density) 2 2.88Mb (extra-high-density) The low three bytes give an indication of the origin of the unit, the value 0x415443 ('ATC') indicates an Atari line-fit or retro-fitted unit.
_FLK	the version number of file sharing extensions. This cookie is installed by a GEMDOS extension which supports the Atari file locking extensions (i.e. the additional Fopen modes and the Flock call). If the cookie cannot be found file sharing is not available.
_FPU	This gives an indication of any floating point unit installed in the machine. Only the high word is used at the time of writing. The bits are used as follows (when set): 0 I/O mapped 68881 (e.g. Atari's SFP004) 1 68881/68882 (unsure which) 2 If bit 1 == 0 then 68881, else 68882 3 68040 internal floating point support
_FRB	'Fast RAM Buffer'. This is used on the TT to give the address of a 64K buffer in ST RAM that all ACSI devices performing DMA can use, when transfers to TT RAM are requested. It is not present if there is no fast RAM.
_IDT	preferred international time and date display mode. The high word is currently unused and is reserved. The low word is broken up into three sections. Bits 15...12 (Time) 0 12 hour clock 1 24 hour clock Bits 11...8 (Date) 0 MM-DD-YY 1 DD-MM-YY 2 means YY-MM-DD 3 means YY-DD MM Bits 7...0 are the ASCII value to be used as a separator, e.g. /. This cookie is used by any AES ≥ 3.30.

_MCH

This gives the machine type; it consists of a minor number (low word) and a major number (high word) as follows:

Major	Minor	Machine
0	0	520/1040 or Mega ST
1	0	STe
1	16	Mega STe
2	0	TT
3	0	Falcon030

Normally you should use the more specific cookies given above, in case some one has added a 68030 processor to an STe, for example.

One possible use for this cookie is to detect the presence of the extra TT serial ports.

_NET

Networking software is available; the value is a pointer to a structure of the following form:

```
struct netinfo
      long publisher_id;
      long version;
};
```

At the time of writing the following `publisher_ids` had been assigned:

Application Design Software `'A&D\0'`
Pams Software `'PAMS'`
Itos Software `'ITOS'`

_SND

This is bit oriented as follows (when the bits are set):

bit 0	ST style GI/Yamaha chip
bit 1	8-bit TT/STe style DMA sound
bit 2	16-bit Falcon030 style CODEC
bit 3	56001 DSP
bit 4	Switch (connection) matrix

_SWI

The STe, TT and Falcon030 have internal configuration switches; this gives their value.

_VDO

the part number of the video shifter:

0x00000000	ST
0x00010000	STe
0x00020000	TT
0x00030000	Falcon030

FSMC	this cookie is inserted by font scaling versions of GDOS; the value is a pointer to a structure of the type:
	`struct {`
	`  long id;  /* '_SPD' */`
	`  short ver;  /* major/minor rev */`
	`  short qual;  /* quality setting */`
	`};`

MiNT	the presence of the MiNT cookie indicates that MiNT is active and that the MiNT extensions are available. At present the high word is unused; the high byte of the low word contains the major version, the low byte the minor version number.
PMMU	if present this cookie indicates that a program has claimed sole access to the machines memory management unit; this occurs for example when using virtual memory managers and MiNT to indicate to MiNT that an MMU table manager is already installed and that memory protection should not be installed.

Note that the absence of a cookie *indicates nothing* about the state of a resource; the host machine may have a 68030, 68882 and high-density floppy fitted with no indication from the cookie jar.

For both Lattice C 5.5 and HiSoft BASIC 2 a ready-made function, `getcookie`, is available for interrogating the cookie jar, for Devpac 3 an example of scanning this list is printed in the manual; you should refer to this, or other, documentation on who to access the cookies.

Appendix D
Language Specific Issues

This section includes details of how the relevant libraries, definitions and declarations for HiSoft BASIC 2, Devpac 3 and Lattice C 5, are included when accessing the functions discussed in this book.

Video Sub-system

HiSoft BASIC 2

When compiling a program which contains operating system calls documented as part of the video sub-system, ensure that the program contains the following line before any of the XBIOS routines are called:

```
LIBRARY "FALCON"
```

Devpac 3

When assembling a program which contains operating system calls documented as part of the video sub-system, ensure that the program contains the following line somewhere within your program:

```
INCLUDE XBIOS.I
```

This file contains the necessary equates to enable the machine code programmer to use the calls by name rather than by its less obvious call number.

Lattice C 5

When compiling a program which contains operating system calls documented as part of the video sub-system, ensure that the program contains the following lines somewhere near the top of your program:

```
#include <osbind.h>
```

These files contain the necessary bindings required by Lattice C to compile correctly. In addition the header file mode.h includes definitions of many of the constants which can be passed to calls in the video subsystem.

Audio Sub-system

HiSoft BASIC 2

When compiling a program which contains operating system calls documented as part of the audio sub-system, ensure that the program contains the following line somewhere near the top of your program:

```
LIBRARY "FALCON"
```

Devpac 3

When assembling a program which contains operating system calls documented as part of the audio sub-system, ensure that the program contains the following line somewhere within your program:

```
INCLUDE XBIOS.I
```

This file contains the necessary equates which allow the programmer to access the calls by name rather than by their operating system numbers.

Lattice C 5

When compiling a program which contains operating system calls documented as part of the audio sub-system, ensure that the program contains the following line somewhere near the top of your program:

```
#include <osbind.h>
```

This file contains the definitions and code required by Lattice C to compile correctly. In addition the header file sndbind.h includes definitions of many of the constants which can be passed to calls in the sound subsystem.

DSP Sub-system

HiSoft BASIC 2

When compiling a program which contains operating system calls documented as part of the DSP sub-system, ensure that the program contains the following line somewhere near the top of your program:

```
LIBRARY "FALCON"
```

Devpac 3

When assembling a program which contains operating system calls documented as part of the DSP sub-system, ensure that the program contains the following line somewhere within your program:

```
INCLUDE XBIOS.I
```

This file contains the necessary definitions, macros and machine code to execute the calls.

Lattice C 5

When compiling a program which contains operating system calls documented as part of the DSP sub-system, ensure that the program contains the following line somewhere near the top of your program:

```
#include <osbind.h>
```

This file contains the definitions and code required by Lattice C to compile correctly. In addition the header file dspbind.h includes definitions of constants and structures which can be passed to calls in the DSP subsystem.

GEMDOS/MiNT

HiSoft BASIC 2

When compiling a program which contains operating system calls documented as part of the MiNT section, ensure that the program contains the following line somewhere near the top of your program:

```
LIBRARY "MINT"
```

Note that BASIC routines which require a string parameter a$ should be passed as SADD(a$ + chr$(0)).

Devpac 3

When assembling a program which contains operating system calls documented as part of the MiNT section, ensure that the program contains the following line somewhere within your program:

```
INCLUDE     GEMDOS.I
```

This file contains the necessary definitions, macros and machine code to make calls into MiNT via TRAP #1.

Lattice C 5

When compiling a program which contains operating system calls documented as part of the MiNT section, ensure that the program contains the following line somewhere near the top of your program:

```
#include <mintbind.h>
```

This file contains the definitions and code required by Lattice C to compile correctly.

AES Enhancements

HiSoft BASIC 2

When compiling a program which contains operating system calls documented in the AES enhancements section, ensure that the program contains the following line somewhere near the top of your program:

```
LIBRARY "GEMAES"
```

This will ensure that the relevant BASIC command extensions will be parsed at compile time and that the necessary code is linked in to the final executable program.

Devpac 3

When assembling a program which contains operating system calls documented in the AES enhancements section, ensure that the program contains the following line somewhere near the start of your program:

```
INCLUDE     GEMMACRO.I
```

This file contains the necessary definitions, macros and machine code to execute the calls in both executable or linkable assembly code. In the case of executable assembly, it will be further necessary to include the following line into the code, usually near the end of the program:

```
INCLUDE     AESLIB.S
```

For any further AES linking and assembly details, please refer to the GEM LIBRARIES section of the Devpac user manual.

Note that in this section the Devpac calling syntax provides a list of the parameters which must be passed to the call. The command must be typed into a machine code listing as it is written here, but omitting the size specifications (.W and .L, which are include for reference only). The assembler's macro facilities automatically expand the command name and any relevant parameters into the full AES calling code. Any data which is passed back out of the call as a result will be placed in the relevant AES output arrays, these are documented where necessary. In the case of an input parameter being a pointer to some data, it is of course quite possible that a result may also be passed back at this address.

Lattice C 5

When compiling a program which contains operating system calls documented in the AES enhancements section, ensure that the program contains the following line somewhere near the top of your program:

```
#include <aes.h>
```

This file contains the definitions required by Lattice C to compile correctly. When linking code which includes some AES calls, it may be necessary to instruct the compiler (either integrated or stand-alone) to build a GEM application, either by use of the Build GEM Application box when integrated or by the -Lg when stand-alone.

SpeedoGDOS

BASIC 2

When compiling a program which contains operating system calls documented as part of the SpeedoGDOS section, ensure that the program contains the following line somewhere near the top of your program:

```
LIBRARY "SPEEDO"
```

Devpac 3

When assembling a program which contains operating system calls documented as part of the SpeedoGDOS section, ensure that the program contains the following line somewhere near the start of your program:

```
INCLUDE GEMMACRO.I
```

This file contains the necessary definitions, macros and machine code to execute the calls in both executable or linkable assembly code. In the case of executable assembly, it will be further necessary to include the following line into the code, usually near the end of the program:

```
INCLUDE VDILIB.S
```

For any further VDI linking and assembly details, please refer to the GEM libraries section of the Devpac user manual.

Note that in this section the Devpac calling syntax provides a list of the parameters which must be passed to the call. The command must be typed into a machine code listing as it is written here. The assemblers macro facilities automatically expand the command name and any relevant parameters into the full VDI calling code. Any data which is passed back out of the call as a result will be placed in the relevant VDI output arrays, these are documented where necessary. In the case of an input parameter being a pointer to some data, it is of course quite possible that a result may also be passed back at this address.

Speedo and strings

Some of the Speedo calls return a device string in a buffer. This string takes the form of the ASCII characters which go to make up the device name, one character in each element of the intout array. That is to say that each character is passed back in a word format with the most significant byte of each word padded out with $00. Before use, the programmer will probably have to unpack the string into a separate storage area or buffer.

Lattice C

When compiling a program which contains operating system calls documented as part of the SpeedoGDOS section, ensure that the program contains the following line somewhere near the top of your program:

```
#include <vdi.h>
```

This file contains the definitions required by Lattice C to compile correctly. When linking code which includes some VDI calls, it may be necessary to instruct the compiler (either integrated or stand-alone) to build a GEM application, either by use of the Build GEM Application box when integrated or by the -Lg when stand-alone.

For all C bindings the VDI workstation handle is explicitly passed to every call (unlike for Devpac/BASIC which use an implicit internal handle); this handle is obtained in the normal way via v_opnvwk/v_opnwk.

Appendix E
OS Binding Numbers

For those readers using an older language which does not yet have support for the newer OS functions, or those attempting to 'roll their own' bindings, a complete list of OS binding 'numbers' are show below:

AES Opcode Numbers

The following table shows the opcodes which the AES uses for its functions:

appl_init	$0a	form_center	$36
appl_read	$0b	form_keybd	$37
appl_write	$0c	form_button	$38
appl_find	$0d	graf_rubberbox	$46
appl_tplay	$0e	graf_dragbox	$47
appl_trecord	$0f	graf_movebox	$48
appl_search	$12	graf_growbox	$49
appl_exit	$13	graf_shrinkbox	$4a
appl_getinfo	$82	graf_watchbox	$4b
evnt_keybd	$14	graf_slidebox	$4c
evnt_button	$15	graf_handle	$4d
evnt_mouse	$16	graf_mouse	$4e
evnt_mesag	$17	graf_mkstate	$4f
evnt_timer	$18	scrp_read	$50
evnt_multi	$19	scrp_write	$51
evnt_dclick	$1a	fsel_input	$5a
menu_bar	$1e	fsel_exinput	$5b
menu_icheck	$1f	wind_create	$64
menu_ienable	$20	wind_open	$65
menu_tnormal	$21	wind_close	$66
menu_text	$22	wind_delete	$67

menu_register	$23	wind_get	$68
menu_popup	$24	wind_set	$69
menu_attach	$25	wind_find	$6a
menu_istart	$26	wind_update	$6b
menu_settings	$27	wind_calc	$6c
objc_add	$28	wind_new	$6d
objc_delete	$29	rsrc_load	$6e
objc_draw	$2a	rsrc_free	$6f
objc_find	$2b	rsrc_gaddr	$70
objc_offset	$2c	rsrc_saddr	$71
objc_order	$2d	rsrc_obfix	$72
objc_edit	$2e	rsrc_rcfix	$73
objc_change	$2f	shel_read	$78
objc_sysvar	$30	shel_write	$79
form_do	$32	shel_get	$7a
form_dial	$33	shel_put	$7b
form_alert	$34	shel_find	$7c
form_error	$35	shel_envrn	$7d

VDI Opcode Numbers

The following table shows the opcodes which the VDI uses for its functions; where two numbers are show separated by a dash the second number is the VDI sub-opcode (placed in `control[5]`):

v_opnwk	1	vsf_style	24
v_clswk	2	vsf_color	25
v_clrwk	3	vq_color	26
v_updwk	4	vq_cellarray	27
vq_chcells	5-1	vrq_locator	28
v_eeol	5-10	vsm_locator	28
vm_filename	5-100	vrq_valuator	29
vs_curaddress	5-11	vsm_valuator	29
v_curtext	5-12	vrq_choice	30
v_rvon	5-13	vsm_choice	30
v_rvoff	5-14	vrq_string	31
vq_curaddress	5-15	vsm_string	31

v_bar	11-1	vqt_font_info	131
v_arc	11-2	vqt_fontheader	232
v_pieslice	11-3	vqt_trackkern	234
v_circle	11-4	vqt_pairkern	235
v_ellipse	11-5	vst_charmap	236
v_ellarc	11-6	vst_kern	237
v_ellpie	11-7	v_getbitmap_info	239
v_rbox	11-8	vqt_f_extent	240
v_rfbox	11-9	v_ftext	241
v_justified	11-10	v_getoutline	243
v_bez_con	11-13	vst_scratch	244
vst_height	12	vst_error	245
vst_rotation	13	vst_arbpt	246
vs_color	14	vqt_advance	247
vsl_type	15	vqt_devinfo	248
vsl_width	16	v_savecache	249
vsl_color	17	v_loadcache	250
vsm_type	18	v_flushcache	251
vsm height	19	vst_setsize	252
vsm color	20	vst_skew	253
vst_font	21	vqt_get_table	254
vst color	22	vqt_cachesize	255
vsf_interior	23	v_set_app_buff	-1

GEMDOS/MiNT Binding Numbers

The following tables show the binding numbers when interfacing to GEMDOS or MiNT via TRAP #1.

GEMDOS

Pterm0	$0	Ptermres	$31
Cconin	$1	Dfree	$36
Cconout	$2	Dcreate	$39
Cauxin	$3	Ddelete	$3a

Cauxout	$4	Dsetpath	$3b
Cprnout	$5	Fcreate	$3c
Crawio	$6	Fopen	$3d
Crawcin	$7	Fclose	$3e
Cnecin	$8	Fread	$3f
Cconws	$9	Fwrite	$40
Cconrs	$a	Fdelete	$41
Cconis	$b	Fseek	$42
Dsetdrv	$e	Fattrib	$43
Cconos	$10	Mxalloc	$44
Cprnos	$11	Fdup	$45
Cauxis	$12	Fforce	$46
Cauxos	$13	Dgetpath	$47
Maddalt	$14	Malloc	$48
Srealloc	$15	Mfree	$49
Dgetdrv	$19	Mshrink	$4a
Fsetdta	$1a	Pexec	$4b
Super	$20	Pterm	$4c
Tgetdate	$2a	Fsfirst	$4e
Tsetdate	$2b	Fsnext	$4f
Tgettime	$2c	Frename	$56
Tsettime	$2d	Fdatime	$57
Fgetdta	$2f	Flock	$5c
Sversion	$30		

MiNT

Syield	$ff	Talarm	$120
Fpipe	$100	Pause	$121
Fcntl	$104	Sysconf	$122
Finstat	$105	Psigpending	$123
Foutstat	$106	Dpathconf	$124
Fgetchar	$107	Pmsg	$125
Fputchar	$108	Fmidipipe	$126
Pwait	$109	Prenice	$127
Pnice	$10a	Dopendir	$128
Pgetpid	$10b	Dreaddir	$129

Pgetppid	$10c	Drewinddir	$12a
Pgetpgrp	$10d	Dclosedir	$12b
Psetpgrp	$10e	Fxattr	$12c
Pgetuid	$10f	Flink	$12d
Psetuid	$110	Fsymlink	$12e
Pkill	$111	Freadlink	$12f
Psignal	$112	Dcntl	$130
Pvfork	$113	Fchown	$131
Pgetgid	$114	Fchmod	$132
Psetgid	$115	Pumask	$133
Psigblock	$116	Psemaphore	$134
Psigsetmask	$117	Dlock	$135
Pusrval	$118	Psigpause	$136
Pdomain	$119	Psigaction	$137
Psigreturn	$11a	Pgeteuid	$138
Pfork	$11b	Pgetegid	$139
Pwait3	$11c	Pwaitpid	$13a
Fselect	$11d	Dgetcwd	$13b
Prusage	$11e	Salert	$13c
Psetlimit	$11f		

BIOS Binding Numbers

The following tables show the binding numbers when interfacing to the Atari BIOS via TRAP #13:

Getmpb	0	Tickcal	6
Bconstat	1	Getbpb	7
Bconin	2	Bcostat	8
Bconout	3	Mediach	9
Rwabs	4	Drvmap	10
Setexc	5	Kbshift	11

XBIOS Binding Numbers

The following tables show the binding numbers when interfacing to the Atari XBIOS via TRAP #14:

Initmous	0	EsetBank	82
Ssbrk	1	EsetColor	83
Physbase	2	EsetPalette	84
Logbase	3	EgetPalette	85
Getrez	4	EsetGray	86
Setscreen	5	EsetSmear	87
Setpalette	6	Vsetmode	88
Setcolor	7	Montype	89
Floprd	8	VsetSync	90
Flopwr	9	VgetSize	91
Flopfmt	10	VsetRGB	93
Midiws	12	VgetRGB	94
Mfpint	13	Dsp_DoBlock	96
Iorec	14	Dsp_BlkHandShake	97
Rsconf	15	Dsp_BlkUnpacked	98
Keytbl	16	Dsp_InStream	99
Random	17	Dsp_OutStream	100
Protobt	18	Dsp_IOStream	101
Flopver	19	Dsp_RemoveInterrupts	102
Scrdmp	20	Dsp_GetWordSize	103
Cursconf	21	Dsp_Lock	104
Settime	22	Dsp_Unlock	105
Gettime	23	Dsp_Available	106
Bioskeys	24	Dsp_Reserve	107
Ikbdws	25	Dsp_LoadProg	108
Jdisint	26	Dsp_ExecProg	109
Jenabint	27	Dsp_ExecBoot	110
Giaccess	28	Dsp_LodToBinary	111
Offgibit	29	Dsp_TriggerHC	112
Ongibit	30	Dsp_RequestUniqueAbility	113
Xbtimer	31	Dsp_GetProgAbility	114

Appendix F
The SpeedoGDOS Font Header

Every SpeedoGDOS font includes a header which gives information about the font; this information may be retrieved by the `vqt_fontheader` call. The following is a list of offsets, names (contained in SPEEDOHD.H for Lattice C) and meanings for this data:

Name	offset	Description
FH_FMVER	0	ASCII font manager version code
		'D4.0' + CR + LF + NULL + NULL - 8 bytes
FH_FNTSZ	8	Font size (bytes) - 4 bytes
FH_FBFSZ	12	Min font buffer size (bytes) - 4 bytes
FH_CBFSZ	16	Min char buffer size (bytes) - 2 bytes
FH_HEDSZ	18	Header size (bytes) - 2 bytes
FH_FNTID	20	Source Font ID - 2 bytes
FH_SFVNR	22	Source Font Version Number - 2 bytes
FH_FNTNM	24	Source Font Name - 70 bytes
FH_MDATE	94	Manufacturing Date - 10 bytes
FH_LAYNM	104	Layout Name - 70 bytes
FH_CPYRT	174	Copyright Notice - 78 bytes
FH_NCHRL	252	Number of Chars in Layout - 2 bytes
FH_NCHRF	254	Total Number of Chars in Font - 2 bytes
FH_FCHRF	256	Index of first char in Font - 2 bytes
FH_NKTKS	258	Number of kerning tracks in font - 2 bytes
FH_NKPRS	260	Number of kerning pairs in font - 2 bytes

FH_FLAGS	262	Font flags - 1 byte:
		Bit 0 Extended font Bit 1 not used Bit 2 not used Bit 3 not used Bit 4 not used Bit 5 not used Bit 6 not used Bit 7 not used
FH_CLFGS	263	Classification flags - 1 byte:
		Bit 0 Italic Bit 1 Monospace Bit 2 Serif Bit 3 Display Bit 4 not used Bit 5 not used Bit 6 not used Bit 7 not used
FH_FAMCL	264	Family Classification value - 1 byte:
		0 Don't care 1 Serif 2 Sans serif 3 Monospace 4 Script or calligraphic 5 Decorative 6-255 not used

FH_FRMCL	265	Font form Classification - 1 byte: Bits 0-3 (width type): 0-3 not used 4 Condensed 5 not used 6 Semi-condensed 7 not used 8 Normal 9 not used 10 Semi-expanded 11 not used 12 Expanded 13-15 not used Bits 4-7 (Weight): 0 not used 1 Thin 2 Ultralight 3 Extralight 4 Light 5 Book 6 Normal 7 Medium 8: Semibold 9: Demibold 10 Bold 11 Extrabold 12 Ultrabold 13 Heavy 14 Black 15 not used
FH_SFNTN	266	Short Font Name - 32 bytes
FH_SFACN	298	Short Face Name - 16 bytes
FH_FNTFM	314	Font form - 14 bytes
FH_ITANG	328	Italic angle - 2 bytes (1/256th degree)
FH_ORUPM	330	Number of ORUs per em - 2 bytes
FH_WDWTH	332	Width of Wordspace - 2 bytes
FH_EMWTH	334	Width of Emspace - 2 bytes
FH_ENWTH	336	Width of Enspace - 2 bytes
FH_TNWTH	338	Width of Thinspace - 2 bytes
FH_FGWTH	340	Width of Figspace - 2 bytes
FH_FXMIN	342	Font-wide min X value - 2 bytes
FH_FYMIN	344	Font-wide min Y value - 2 bytes
FH_FXMAX	346	Font-wide max X value - 2 bytes
FH_FYMAX	348	Font-wide max Y value - 2 bytes

FH_ULPOS	350	Underline position - 2 bytes
FH_ULTHK	352	Underline thickness - 2 bytes
FH_SMCTR	354	Small caps transformation - 6 bytes
FH_DPSTR	360	Display sups transformation - 6 bytes
FH_FNSTR	366	Footnote sups transformation - 6 bytes
FH_ALSTR	372	Alpha sups transformation - 6 bytes
FH_CMITR	378	Chemical infs transformation - 6 bytes
FH_SNMTR	384	Small nums transformation - 6 bytes
FH_SDNTR	390	Small denoms transformation - 6 bytes
FH_MNMTR	396	Medium nums transformation - 6 bytes
FH_MDNTR	402	Medium denoms transformation - 6 bytes
FH_LNMTR	408	Large nums transformation - 6 bytes
FH_LDNTR	414	Large denoms transformation - 6 bytes

The transformation data format is as follows:

Y position	2 bytes
X scale	2 bytes (1/4096ths)
Y scale	2 bytes (1/4096ths)

Appendix G
MultiTOS Configuration

MINT.CNF and GEM.CNF are control files used by MiNT and the multitasking AES respectively, together they allow you to control many aspects of MultiTOS's behaviour. Each file is a plain text file (i.e. they can be edited with a normal text editor), with one command per-line and comments indicated by lines starting with a #.

MiNT Commands and Variables

Variables

The following variables can be set in the MINT.CNF file, the variable names must be all *upper* case and the = and `value` *must* be concatenated with no intervening spaces:

`GEM=<file> <tail>` `INIT=<file> <tail>`	set the full path+name of the file that contains the version of GEM to execute. A command tail may optionally be used with this variable which is also passed to the program executed. The difference between the `GEM=` and the `INIT=` varieties are that when specified via the `GEM=` method the program is started as if via the `exec_os` vector; this can help certain GEM patching programs to install correctly.
`MAXMEM=<kbytes>`	gives the maximum amount of memory that any process may use (in kilobytes). The default is to make this unlimited, but if you have a lot of memory and/or programs that grab more memory than they should, try setting this.

SLICES=<n>	controls how long a process may run before being interrupted. The default value (2) is usually best, but if you tend to run very processor intensive applications in the foreground, you might want to put SLICES=3 (this makes CPU hogs get more time than they otherwise would).
CON=<file>	specify initial file/device for handles -1, 0, 1
PRN=<file>	specify initial file for handle 3
BIOSBUF=[yn]	if n or N then turn off the BIOS buffering; by default MiNT buffers up BIOS output (to improve performance); BIOSBUF=n disables this feature.
DEBUG_LEVEL=<n>	set debug level to (decimal number) n; this controls output of debugging information, the higher the level, the more stuff MiNT will generate about what it's doing. The average user doesn't want to hear about this stuff, so the default is 0.
DEBUG_DEVNO=<n>	set debug device number to (decimal number) n; this is the BIOS device number to which the debug info should be sent.

Note that the BIOSBUF, DEBUG_LEVEL and DEBUG_DEVNO variables should be considered as 'development' settings and so may not be available in future versions of the kernel.

Commands

The following commands can be used in the MINT.CNF file, command names must be all *lower* case with a single space between any command and its argument:

echo message	print a message on the screen
alias drive path	make a fake drive pointing at a path
cd dir	change directory/drive
exec cmd args	execute a program
setenv name val	set up environment
sln file1 file2	create a symbolic link
ren file1 file2	rename a file

GEM Commands and Variables

Variables

The following variables can be set in the GEM.CNF file, the variable names must be all *upper* case and the = and `value` *must* be concatenated with no intervening spaces:

AE_FONTID=	The font ID of the font which is to be used as the system font; if this is not specified the default system font (1) is used.
AE_PNTSIZE=	The point size of the font which is to be used as the system font; if this is not specified the default point size (13) is used.
AE_SREDRAW=	The AES normally sends a full-screen redraw message when a GEM program starts up (calls `appl_init`); if this variable is set to zero then this message is not sent.
AE_TREDRAW=	The AES normally sends a full-screen redraw message when a GEM program finishes (calls `appl_exit`); if this variable is set to zero then this message is not sent.

Commands

The following commands can be used in the GEM.CNF file, command names must be all *lower* case with a single space between any command and its argument:

`run cmd`	execute a program
`setenv name=val`	set up environment

Environment Variables

The following environment variables, which can be set either in the MINT.CNF or the GEM.CNF file, are used by the AES to control its behaviour:

ACCEXT	a comma-separated list of extensions which are to be considered accessories.

ACCPATH	a comma-separated list of directories which will be searched for accessories at startup time. When an accessory is found in a given directory, that directory will be the accessory's default directory when it starts. The root directory of the boot device is always searched in addition to any directories appearing in ACCPATH.
DESKCOPY	the full pathname of the program which is run by the Desktop for file copies, moves and renames; a discussion of the command tail passed by the Desktop is shown below.
DESKFMT	the full pathname of the program which is run by the Desktop for disk copies and formats; a discussion of the command tail passed by the Desktop is shown below.
GEMEXT	a comma-separated list of extensions which are to be considered GEM programs.
PATH	a comma-separated list of directories which will be searched for programs when shel_write is called in mode 0, 1 or 3. In addition, `shel_find` and `rsrc_load` will look in all directories in this path when searching for files.
SHPRINT	the full pathname of the program which is run by the Desktop for printing files; the pathname of the file to be printed is passed in the command tail to the program.
SHSHOW	the full pathname of the program which is run by the Desktop for showing files; the pathname of the file to be displayed is passed in the command tail to the program.
TOSEXT	a comma-separated list of extensions which are to be considered TOS programs.
TOSRUN	when starting a program the AES looks for the environment variable TOSRUN which should contain the full path of a TOS handler program, to which the AES will pass the TOS program name into the command tail.

When launching the program specified by DESKFMT, a command tail of the form shown below is used, if a format operation is required:

```
-f A:
```

If a disk-copy operation is being performed the command tail is of the form:

```
-c A: B:
```

indicating a copy from the disk in drive `A:` to drive `B:`.

When launching the program specified by `DESKCOPY`, a command tail of the form shown below is used, if a copy operation is required:

```
-c [-options ...] [files names ...] [destination path]
```

for a file deletion:

```
-d [-options ...] [files names ...]
```

for a file move:

```
-m [-options ...] [files names ...] [destination path]
```

The following option letters may also be generated by the Desktop to indicate the current Desktop preferences:

File copy confirmation:	-A	Yes
	-B	No
File deletion confirmation	-C	Yes
	-D	No
File overwrite confirmation	-E	Yes
	-F	No
Destination renaming	-R	

Appendix H
Signals and Error Codes

This appendix outlines various handles defined by Atari within the MiNT system. These break down into the error codes which are returned by MiNT calls and the signals which are passed back and forth between concurrent applications. Extensive use of these abbreviations are made in the relevant documentation elsewhere in this manual.

BIOS error codes:

ERROR	-1	generic error
EDRVNR	-2	drive not ready
EUNCMD	-3	unknown command
E_CRC	-4	CRC error
EBADRQ	-5	bad request
E_SEEK	-6	seek error
EMEDIA	-7	unknown media
ESECNF	-8	sector not found
EPAPER	-9	out of paper
EWRITF	-10	write fault
EREADF	-11	read fault
EWRPRO	-13	device write protected
E_CHNG	-14	media change detected
EUNDEV	-15	unknown device
EBADSF	-16	bad sectors on format
EOTHER	-17	insert other disk request

GEMDOS/MiNT error codes

EINVFN	-32	invalid function
EFILNF	-33	file not found
EPTHNF	-34	path not found
ENHNDL	-35	no more handles
EACCDN	-36	access denied
EIHNDL	-37	invalid handle
ENSMEM	-39	insufficient memory
EIMBA	-40	invalid memory block address
EDRIVE	-46	invalid drive specification
EXDEV	-48	cross device rename
ENMFIL	-49	no more files (from Fsnext)
ELOCKED	-58	record is locked already
ENSLOCK	-59	invalid lock removal request
ERANGE	-64	range error
ENAMETOOLONG	-64	a filename component is too long
EINTRN	-65	internal error
EPLFMT	-66	invalid program load format
ENOEXEC	-66	as above
EGSBF	-67	memory block growth failure
ELOOP	-80	too many symbolic links

MiNT signals

SIGNULL	0	not really a signal
SIGHUP	1	hangup signal
SIGINT	2	sent by ^C
SIGQUIT	3	quit signal
SIGILL	4	illegal instruction
SIGTRAP	5	trace trap
SIGABRT	6	abort signal
SIGPRIV	7	privilege violation
SIGFPE	8	divide by zero
SIGKILL	9	cannot be ignored

SIGBUS	10	bus error
SIGSEGV	11	illegal memory reference
SIGSYS	12	bad argument to a system call
SIGPIPE	13	broken pipe
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16	urgent condition on I/O channel
SIGSTOP	17	stop signal not from terminal
SIGTSTP	18	stop signal from terminal
SIGCONT	19	continue stopped process
SIGCHLD	20	child stopped or exited
SIGTTIN	21	read by background process
SIGTTOU	22	write by background process
SIGIO	23	I/O possible on a descriptor
SIGXCPU	24	CPU time exhausted
SIGXFSZ	25	file size limited exceeded
SIGVTALRM	26	virtual timer alarm
SIGPROF	27	profiling timer expired
SIGWINCH	28	window size changed
SIGUSR1	29	user signal 1
SIGUSR2	30	user signal 2
Handler names:		
SIG_DFL	0	default signal action
SIG_IGN	1	ignore signal action

Appendix I
Bibliography

This bibliography contains our suggestions for further reading on the subject of the Atari's operating system extensions and 680x0 and assembly language. The views expressed are our own and as with all reference books there is no substitute for looking at the books in a good bookshop before making a decision.

Atari Falcon030 Developer Documentation
Atari Corp. [1992]

Atari Corp., 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

DSP56000/DSP56001 Digital Signal Processor User's Manual
Motorola Inc. [1990]

DSP56000UM/AD Rev.2, Motorola Literature Distribution, P.O. Box 20912 Phoenix, AZ 85036, USA.

M68000 Family Programmer's Reference Manual
Motorola Inc. [1992]

M68000PM/AD Rev.1, Motorola Literature Distribution, P.O. Box 20912 Phoenix, AZ 85036, USA.

MC68030 32-Bit Microprocessors User's Manual
Motorola Inc. [1987]

Motorola Literature Distribution, P.O. Box 20912 Phoenix, AZ 85036, USA.

MC68040 Microprocessors User's Manual
Motorola Inc. [1992]

Motorola Literature Distribution, P.O. Box 20912 Phoenix, AZ 85036, USA.

MC68881/MC68882 FPU User's Manual **Motorola Inc. [1987]**

ISBN 0-13-566936-7, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

MC68851 PMMU User's Manual **Motorola Inc. [1989]**

ISBN 0-13-566993-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Notes

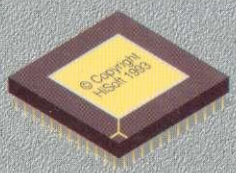
Modern Atari System Software

This reference guide is an invaluable source of documentation for any serious programmer on the Atari 680x0 platform. It contains a wealth of information on using the newer Atari TOS-based operating systems from within C, BASIC and assembly language in addition to many general programming hints and tips.

Topics covered include:

- An overview of the Atari Falcon030 computer including its video, audio and DSP sub-systems and the new system calls needed to handle this exciting hardware
- Details of how to program with MiNT™, MultiTOS™ and SpeedoGDOS™
- Information on using the new features of TOS 4.0, including all the AES enhancements, the Cookie jar, 3D dialogs etc.
- The Atari Style Guide is included which gives advice on programming a consistent user interface so as to improve the look and ease-of-use of your programs
- Technical Appendices cover the SpeedoGDOS™ header, MultiTOS™ configuration, operating system bindings and the various error codes/signals that can be detected

Modern Atari System Software details the interface code for calling all these new functions from Lattice C 5.60, HiSoft BASIC 2.10 and HiSoft Devpac 3.10 and is an essential buy for all Atari developers.



Modern Atari System Software is written and published by HiSoft, Greenfield, UK.

ISBN 0 948517 63 8