

**Cambridge Microprocessor
Systems Limited**

**Minos Real Time
Operating System**

© Cambridge Microprocessor Systems Ltd 1998

The information contained in this document has been thoroughly checked and is believed to be factually correct. However Cambridge Microprocessor Systems Ltd. assumes no responsibility for any mistakes, omissions, or inaccuracies.

Cambridge Microprocessor Systems do not accept any liability out of the use of this or any other Cambridge Microprocessor Systems product.

Although Cambridge Microprocessor Systems Ltd. is committed to supplying its customers a standard, well tested product, we reserve the right to make modifications to the product described in this manual.

This page is intentionally left blank

1. Minos Operating System	1
1.1. Introduction	1
1.2. Accessing Minos	1
2. Minos System Features	3
2.1. Memory Management	3
2.2. Memory Organisation and Initialisation.....	3
2.3. Memory Management Functions	3
2.3.1. Memory Blocks	4
2.3.2. Memory Fragmentation	4
2.3.3. Memory Types.....	4
2.4. Multi-Tasking	5
2.5. Process States	7
2.5.1. Running State	7
2.5.2. Waiting State	7
2.5.3. Sleeping State	8
2.6. Creating New Processes	8
2.7. Minos Module Format	8
2.7.1. Module header start marker	9
2.7.2. Module Checksum.....	9
2.7.3. Module Size.....	10
2.7.4. Module Name Pointer.....	10
2.7.5. Module Execution Entry	10
2.7.6. Module Type.....	10
2.7.7. Module Data	11
2.7.8. Module Header End Marker	11
2.8. Example	11
2.9. Minos System Modules.....	12
2.9.1. The 'Sysinit' Configuration Module.....	12
2.9.2. The 100 Hz System Counter	13
2.9.3. Memory Initialisation Table.....	14
3. Interrupt Handling	15
3.1. Using Interrupts in Applications Programs.....	15
3.2. 68000 Vector Tables.....	16
3.3. The Minos Interrupt Polling System	17

4.	Minos I/O Facilities	17
4.1.	I/O Paths	18
4.2.	Device Name Syntax	18
4.3.	Random Access Files	18
4.4.	Configuring Devices	19
4.5.	The Configuration Tables.....	19
5.	Writing Device Drivers.....	22
5.1.	Introduction	22
5.2.	Internal Structure of a DIT.....	22
5.3.	Creating a New DIT	26
5.4.	Device Drivers.....	28
5.4.1.	Internal Structure of a DVS Block.....	28
5.4.2.	Internal Structure of a Device Driver	32
5.5.	Testing New Drivers.....	41
6.	Minos System Functions.....	43
6.1.	Introduction	43
6.2.	_BackUp.....	45
6.3.	_Chain	46
6.4.	_Close	47
6.5.	_Create	47
6.6.	_Dalloc	48
6.7.	_Death.....	48
6.8.	_Debug.....	49
6.9.	_Delete.....	49
6.10.	_Di	50
6.11.	_Dog	50
6.12.	_Echo	51
6.13.	_Ei	51
6.14.	_Escape.....	52
6.15.	_Exit.....	52
6.16.	_Fix_Mod.....	53
6.17.	_Fsize.....	53
6.18.	_Functions	54
6.19.	_GetPos.....	56
6.20.	_Gettime.....	57

6.21.	_Irq	58
6.22.	_Link	59
6.23.	_Malloc	60
6.24.	_MkData	61
6.25.	_ModDir	61
6.26.	_Open	62
6.27.	_Procs	62
6.28.	_RdTick	63
6.29.	_Read	63
6.30.	_ReadLn	64
6.31.	_Ready	65
6.32.	_Reinit	65
6.33.	_ResTyp	66
6.34.	_SetPos	66
6.35.	_Setime	67
6.36.	_Setup	68
6.37.	_Signal	69
6.38.	_Sleep	70
6.39.	_Trim	71
6.40.	_UnBack	71
6.41.	_UnFix	72
6.42.	_UsrRam	72
6.43.	_Vector	73
6.44.	_Wait	73
6.45.	_Write	74
6.46.	_WrTick	74
7.	Minos Error Codes	75
7.1.	Code 1 _E\$Mfull (error 65536)	75
7.2.	Code 2 _E\$BadSum (error 65537)	75
7.3.	Code 3 _E\$Module (error 65538)	75
7.4.	Code 4 _E\$NoPath (error 65539)	75
7.5.	Code 5 _E\$Path (error 65540)	76
7.6.	Code 6 _E\$Mode (error 65541)	76
7.7.	Code 7 _E\$Esc (error 65542)	76
7.8.	Code 8 _E\$Vector (error 65543)	76
7.9.	Code 9 _E\$No_Irq (error 65544)	76
7.10.	Code 10 _E\$Func (error 65545)	76

7.11.	Code 11 _E\$Ready (error 65546).....	76
7.12.	Code 12 _E\$Found (error 65547).....	77
7.13.	Code 13 _E\$Open (error 65548).....	77
7.14.	Code 14 _E\$NetNum (error 65549).....	77
7.15.	Code 15 _E\$Syntax (error 65550).....	77
7.16.	Code 16 _E\$Known (error 65551).....	77
7.17.	Code 17 _E\$Proc (error 65552).....	77
7.18.	Code 18 _E\$Link (error 65553).....	77
7.19.	Code 19 _E\$NoBack (error 65554).....	78
7.20.	Code 20 _E\$BadMem (error 65555).....	78
7.21.	Code 21 _E\$Size (error 65556).....	78
7.22.	Code 22 _E\$Backed (error 65557).....	78
7.23.	Code 23 _E\$Type (error 65558).....	78
8.	Example Programs	79

1. Minos Operating System

Important

The sections of code contained in this manual are not necessarily written to operate with any software support package supplied as part of a package. Many of them are written to be blown directly into PROM once they have been assembled. Other code segments may not be programs at all but just examples to demonstrate a particular point.

1.1. Introduction

The growing movement in the design of software for embedded controller systems towards high level languages has highlighted the need for a high performance real time operating system. The operating system provides a buffer between the high level language and the system hardware easing software development and aiding software portability. It is not always enough to run the development operating system in the target as many of the important features needed to produce a real time system are not present in systems like MSDOS & UNIX. The Module operating system, Minos, which is supplied as standard in all starter packages, is designed to provide a very high speed environment specially suited to supporting high level languages in real time and to aid software development and portability. This section is designed to explain some of the concepts and features of the Minos operating system and to provide the information required in order to use the system from low level machine code routines.

1.2. Accessing Minos

In order to make the system easily ROMable and position independent the Minos system functions are accessed using the 68000's 'trap #0' instruction. Each function within the system has it's own 16 bit function code which must follow the 'trap #0' instruction. After a call to Minos the execution of the applications program continues at the instruction

after the function code. Parameters for the various system calls are passed using the CPU registers. The format of the parameters returned by Minos system calls is somewhat dependent on the function, however, an error in a function is always signalled by the carry bit being set and an error code being returned in d1. The example below shows how to create a time delay by putting the system to sleep using function number 8.

```
    move.l    #100,d0    Delay in 1/100 second
    trap     #$00       Minos call
    dc.w     $08        Delay function code
```

This access procedure is ideally suited to the use of a 'macro' if your assembler supports them, e.g.

```
Sleep equ      8
Minos macr
    trap     #$00
    dc.w     \0          Use parameter passed
endm
```

The program above can now be rewritten as follows:

```
    move.l    #100,d0
    Minos    Sleep
    bcs     D1a_Error    Do some error routine
```

2. Minos System Features

2.1. Memory Management

Memory management is one of the most important and often under rated facilities of any operating system. Poor memory management leads to applications programs that are difficult to modify, wasteful of RAM and very difficult to PROM. Minos provides a highly advanced memory management system that can handle battery backed memory during power down as well as the more conventional memory allocation and de-allocation features.

2.2. Memory Organisation and Initialisation

The main memory of a Minos system is divided up into one or more 'segments'. Each segment can be either battery backed or volatile and can be up to 2 Mbytes long. Memory segments can start at any address and do not need to be contiguous. When a Minos system is reset the memory management system is re-initialised as follows:

- All the volatile segments are cleared to zero and joined by a linked list,
- An allocation table is then created for each segment,
- The integrity of the allocation table in each of the battery backed segments is then checked. If the table is damaged the segment is initialised in the same way as a volatile segment. If the table is intact the segment is added to the list without creating a new allocation table.

2.3. Memory Management Functions

The memory management system can be used by user programs to gain extra memory for variables, tables, buffers etc. Memory claims can be made at any time and can be of any size.

2.3.1. Memory Blocks

The size of the smallest allocable block is a key parameter in any memory management system. If it is too big there is a risk that significant amounts of memory will be wasted, if it is too small the allocation table begins to take up a significant part of the memory and the risk of fragmentation is increased. The Minos system uses an allocation block of 32 bytes and the size of each memory request is rounded up to the nearest 32 bytes before the actual allocation takes place, e.g. if a claim is made for 1020 bytes then 1024 bytes will actually be allocated.

2.3.2. Memory Fragmentation

Memory fragmentation is the name given to the condition where the allocated areas in a memory segment become small and scattered and break up the free RAM into smaller areas. This will cause the memory allocation function to return a memory full error if a request is made for a block which is larger than the largest contiguous block, even though the total available free RAM may be large enough to satisfy the request. It is not possible to completely prevent memory fragmentation but there are a number of ways of reducing its effect. Always try to make one large memory claim, i.e. if the program has 25 byte sized variables to store, make one 25 byte memory claim not 25 1 byte claims. If the program does need more than one claim try to make all the claims at the start of the program as this is likely to make the areas contiguous. If the program only requires a small RAM area, use the stack instead of claiming more memory. Always ensure that all memory is returned.

2.3.3. Memory Types

The Minos memory management system allows two types of memory: type 0 which is battery backed and type 1 which is volatile. The memory types allow RAM to be taken from different memory segments according to the type of data being stored there. For example stacks and buffers etc. are claimed from the volatile memory while program storage is taken from the battery backed area so that it can be preserved during power down. If all the memory of a particular

type is full or if the hardware does not support the requested type then memory claims will automatically be taken from the other type.

Note:

Memory which is claimed from the battery backed area is still volatile, it must be marked using the `_BackUp` function before it will be preserved.

2.3.3.1. Battery Backed Memory

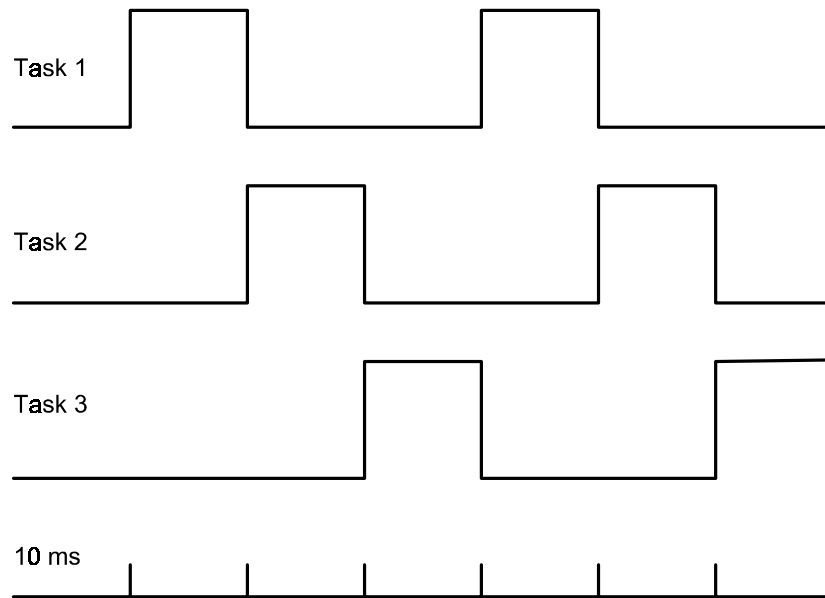
The Minos memory management system is able to maintain all the battery backed memory allocation information even during power failure. To use the battery backed memory a memory claim must be made in type 0 memory. This will allocate an area in the battery backed RAM but will not cause it to be preserved. In order to preserve this block of memory the `_BackUp` function must be used. When a block of battery backed RAM is no longer required it must first be made volatile again with the `_UnBack` function before finally de-allocating it using the `_Dalloc` function.

2.4. Multi-Tasking

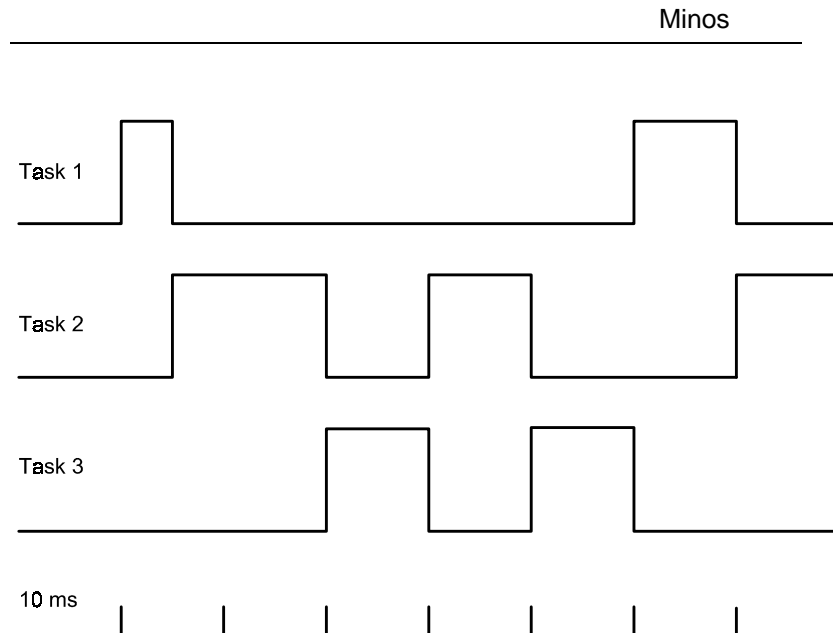
The Minos operating system provides a complete multi-tasking environment that allows a number of different programs, called processes, to be run simultaneously. This is ideally suited to many monitoring and control applications where a number of tasks must be performed in parallel. The Minos operating system also provides a number of inter-process communication facilities, which allow external events or other processes to pass information.

It is the responsibility of the operating system to allocate the resources of the CPU to each process as efficiently as possible in order to maintain a good real time performance. The Minos operating system does this using a round robin process scheduler. This type of process scheduler allocates the resources of the CPU to each process in turn on a fixed time basis (set to 10 ms). The round robin process scheduler forces all processes to have the same priority. Although strict round robin schedulers can be a little restrictive the Minos

scheduler allows tasks to be changed mid way through a time slot and allows the process queue to be reordered, which creates a more 'event driven' structure.



The diagram above shows three tasks all in the running state. Each task is being given a 10 ms slot in strict rotation.



This time task 1 does a 50 ms sleep part way through it's time slot. Notice how task 2 now has a longer slot the first time and how the remaining slots are divided up equally between task 2 and task 3.

2.5. Process States

Each process in a Minos system can exist in one of three states. It can be running, waiting or sleeping.

2.5.1. Running State

The process is active and continues to execute.

2.5.2. Waiting State

Any process can put itself into the waiting state using the `_Wait` system call. This will take the process out of the list of running processes, called the running queue, and place it in a list of waiting processes, called the waiting queue. Processes in the waiting queue do not receive any CPU time at all. They stay in the waiting queue until they are told to rejoin the running queue by a signal from another

process. Programs that are required to take some action when an error condition occurs can be put into the waiting queue and left there until their alarm goes off and they are signalled to continue execution. The process rejoins the running queue so that it will become the active process at the next task change.

2.5.3. Sleeping State

A process can remove itself from the running queue and put itself into the sleeping queue using the `_Sleep` system function. The sleeping state is similar to the waiting state except that the process is returned to the running queue after a fixed time. Time delays can be created without using up the CPU resources. If the process is sent a signal while it is asleep it is woken up early.

2.6. Creating New Processes

New processes can be created in one of two ways either from a Minos program module using the `_Chain` system call or from a 68000 subroutine using `_Fork`. When a new process is created a data structure called a process descriptor is assigned. This is a small block of memory, which contains space for the stack for the new process and some space to save the CPU registers when the process is not running.

2.7. Minos Module Format

One of the biggest problems with stand-alone systems is that every one is different and requires a slightly different configuration. This leads to problems when the time comes to produce PROMs and move from the development to the target environment. Minos eases these problems by using a highly modular structure, which allows programs and hardware drivers to be written as completely autonomous units or 'modules'. These modules are then simply included or left out depending on the configuration. It also means that the Minos system does not need to be modified when new hardware is added you simply add a driver module for that hardware to the application PROM.

Each Minos module has a small header, which contains information about the program, or data that follows it along with a pointer to the name of the module. The 'name' is an ASCII string usually added to the end of the program. These headers are located by the system during its initialisation and the name is added to a list of known modules called the module directory.

The various fields in a Minos module header are shown below.

Name	Offset	Size
Header Start Marker	0	2 bytes (\$4AFC)
Checksum	2	2 bytes
Size	4	4 bytes
Name Pointer	8	4 bytes
Execution Entry	12	4 bytes
Type	16	2 bytes
Data	18	14 bytes
End Marker	32	2 bytes (\$A55A)

2.7.1. Module header start marker

This is a fixed code which is used by Minos to locate the start of a module. It is always set to \$4AFC.

2.7.2. Module Checksum

This is the least significant 16 bits of the ones complement of the sum of all the 16 bit values after the module header. It is used to check that the module is intact before adding it to the module directory.

Note:

The checksum does include the module name.

Sum = (w1 + w2 + w3 + ... +w<size/2>) EOR -1

2.7.3. Module Size

The module size is the number of bytes following the module header including the name. Minos uses it when it calculates the module checksum.

2.7.4. Module Name Pointer

The module name pointer is the offset from the start of the module (the \$4AFC) to an ASCII string terminated by 0, which is the name of the module. Module names can be any length and can include any alphanumeric characters.

2.7.5. Module Execution Entry

The module execution entry is the offset from the start of the module (the \$4AFC) to the start of the program. If the module is a program this will be the first instruction. Some other types of module give a different meaning for the entry point.

2.7.6. Module Type

The module type is used by the Minos system to make sure that a module is capable of performing the requested function before executing it. This ensures that it does not try to run a device driver as a program or make an I/O call to a block of data etc.

The following module types have been defined. The labels are the recommended mnemonics.

_T\$Program	0	68000 Machine Code Program
_T\$Dit	1	Device Initialisation Table
_T\$Driver	2	Device Driver
_T\$System	3	A special system function
_T\$Pcode	4	A Modula-2 Program
_T\$Data	5	A Data Module
_T\$Absolute	6	An Absolute Program
_T\$Subrtn	7	A Subroutine

2.7.7. Module Data

The module data field is used by Minos and some special programs to keep useful system information such as the date when they were created, memory requirements etc. The type of information stored here depends on the type of module. This space is available to users in some modules but this is not recommended.

2.7.8. Module Header End Marker

The module header end marker is used by the system as an added integrity check. It should always be set to \$A55A.

2.8. Example

The following example shows how to get an assembler to add the fields required for a Minos module. The assembler will fill in some of the fields, the rest must be filled in after the program has been assembled.

```
_head dc.w      $4AFC      Module Start marker
      dc.w      0          Checksum
      dc.l      0          Size of the module
      dc.l      0          Module name pointer
      dc.l      _start-_head Module execution
      dc.w      _type      Module type
      dc.l      0          Module data
      dc.l      0          Module data
      dc.l      0          Module data
      dc.w      0          Module data
      dc.w      $A55A      Header end marker

***** End of module header declaration *****
_type      equ      _T$Program
_start     nop          A simple program
           bra.s     _start
```

The assembler will fill in the module type and the execution entry. The rest must be filled in after the program has been assembled.

If you are using an assembler with a linker it is worth turning the module header into a library file and linking it with the make program after it has been assembled.

2.9. Minos System Modules

Minos uses two low level system modules, one to generate regular interrupts for system timing, the other contains all the configuration information for the system.

2.9.1. The 'Sysinit' Configuration Module

The configuration module is called 'sysinit' and contains the names of the devices that will be used by the file server and the terminal. The example below shows a source of the 'sysinit' module used in the development pack PROMs for a particular Module. Each programming environment and controller board will have different set up files.

```
        nam    Sysinit
        use    <mod.d>      Bring in some label
                               definitions
_type   equ    _T$System
_strt   dc.l   Prg_N-_head  Offset to name of first
                               program
        dc.l   Std_N-_head  Offset to name of terminal
        dc.l   Tim_N-_head  Offset to name of 100 Hz
                               Module
        dc.l   Dsk_N-_head  Offset to name of file
                               Server

Prg_N   dc.b   "shell",0
Std_N   dc.b   ":TERM",0
Tim_N   dc.b   "ticker",0
Dsk_N   dc.b   ":D1",0
```

There are four fields in the 'sysinit' file each of them is an offset from the start of the module (the \$4AFC) to the name of another module in ASCII.

2.9.1.1. The Initial Module

This is the name of the module that will be run as the first program. This is set to 'shell' for most C development systems and M2 for Modula-2 systems, so that the board will power up running the correct shell.

2.9.1.2. The Initial Terminal

This is the name of the device that gets assigned to the standard terminal (path 0). In the development system it is set to ":TERM" which always refers to the serial port being used as the system terminal.

2.9.1.3. The Clock Name

This is the name of the module that provides the timing reference for the Minos system.

2.9.1.4. The Initial File Server

This is the name of the device that is used to provide the default file server. The example above assumes that it will be linked with a library file that contains the module header.

Note:

All of these four fields must be filled in even if the application does not use the terminal or file server. The four modules that the names refer to must be included in the PROMs.

2.9.2. The 100 Hz System Counter

In order to provide accurate time delays Minos keeps a 100 Hz elapsed time counter which is driven by a hardware timer. This counter can be read or written at any time allowing intervals to be timed with a fixed error of 0–20 ms. A second counter which decrements to zero is used to create delays.

2.9.3. Memory Initialisation Table

This data block can be found immediately after the MINOS operating system code in the PROM on the controller. It is used to define the memory areas for the controller. The first four bytes in this file are all \$ff to indicate the start of a new memory type. The next four bytes define the start address of the first block of static RAM followed by the end address of the first block of static RAM. Any extra blocks are included after this. Then there is four bytes of \$00 and to signify the end of this memory type. Four bytes of \$ff indicate the start of the PROM block. The next four bytes define the start address of the PROM and then the end address of the PROM. Like the RAM this can be followed by more blocks if required. The whole initialisation table is completed with four bytes of \$00.

3. Interrupt Handling

One of the most important features of a real time system is its ability to cope with external interrupts. Much of the very low level polling and memory control is taken care of by Minos allowing interrupt service routines to be written as subroutines which are called in turn as part of the service for each interrupt level. When an interrupt occurs the system saves some working registers, picks up a pointer to the systems global variables and sets the value of a2 back to what it was when the service routine was installed. It then jumps to the first service routine as if it was a subroutine, with the registers set as follows:

- a6 Pointing to the start of the system variables
- a2 Has the same value as when the service routine was installed
 (usually pointing at some RAM for the service routine)

Registers d0, d1 and a1 to a3 can be used without preserving them first. Any other registers that are used should be preserved. If a service routine finds that its hardware was not responsible for the interrupt it should return with the carry bit set in the status register. The service routines should return with an RTS instruction not an RTE and with the carry clear.

3.1. Using Interrupts in Applications Programs

Hardware interrupts provide an efficient way of responding to external events that require a very prompt response time or have a high data rate such as serial ports, alarm switches etc. Under normal circumstances, where the required response time is long (greater than the number of processes * the time slot time), a new process can be created using a high level language. These processes can be run under control of the operating system to monitor the events. A good operating system should also provide all the interrupt handling required by the serial ports, keyboards and provide a nicely buffered data stream for the applications program. Some applications however do require a very fast response to external events or errors. In this

case a hardware interrupt is ideal. These interrupts can be serviced either by a new device driver (which will be described later) or by integrating an interrupt service routine with the application program. The rest of this section describes the interrupt facilities provided by the Minos operating system and how they can be used from an application program.

3.2. 68000 Vector Tables

The 68000 expects to find all its interrupt and exception vectors in a table starting at 0. (Full details of the 68000 vector table and vector numbers may be found in a 68000 data sheet). Since this table also contains the reset vector for the CPU it is likely that this table will be in PROM making it impossible to install new interrupt handlers without rebuilding the PROMs. In order to allow new interrupt services to be installed while the system is running, the Minos operating system provides a short service for every interrupt which uses a second vector table in RAM to reach the real interrupt service routine. The order of the RAM vector table is exactly the same as that of a real vector table and the code used to access it is as follows:

```
move.l      VectorBase + Vector * 4, (a7)
rts
```

This adds about 40 CPU clock cycles to the interrupt overhead but greatly eases the development of interrupt driven applications. The base address of the RAM vector table can be located with the Minos `_Vectors` function. Routines which require very fast response times can be installed by placing the address of the service routine in the RAM vector table before enabling the interrupt source. Routines that are installed here must preserve all the CPU registers and return with an 'rte' instruction. Although this method provides the fastest response time the service routine must take care of all possible interrupt sources using that vector.

3.3. The Minos Interrupt Polling System

The Minos operating system provides a general interrupt service routine which will call a number of potential service routines as subroutines until it finds one that can service this interrupt. This routine also keeps a workspace pointer for each of the service routines, which can be used to pass the address of a parameter block between the application program and the service, routine. The polling system adds about 400 clock cycles to the interrupt overhead but this is still small compared with the typical time of a service routine. Service routines that are to be installed into the polling table should be written as subroutines, i.e. they must return with an 'rts' not an 'rte'. If the routine does manage to service the interrupt it should return with the carry bit clear otherwise it should return with carry set. Registers d0, d1, a1, a2, and a3 may be used without saving them.

4. Minos I/O Facilities

One of the most important features of any target operating system is its ability to handle the reading and writing of characters to and from serial ports, graphics displays etc. Minos provides a number of read, write and status calls which provide a completely hardware independent I/O structure which is ideal for supporting high level languages. The systems I/O routines allow access to the various pieces of hardware on the system by means of a group of well defined subroutines called a device driver. These offer two very useful features. Firstly, a new piece of hardware can be added by simply writing a driver for that hardware which conforms to the Minos driver specification. Secondly, application programs can be written without any knowledge of the hardware that they will be using for their input and output. Minos identifies the devices being used by means of a number of paths. A path is a small data structure that is created each time the application program tries to open a device. Application programs keep a path number for each of the paths being used and use that path number to specify the I/O device to be used when I/O calls are made.

4.1. I/O Paths

Application programs need to specify a 'path number' when they call a Minos I/O function. The `_Create` or `_Open` function can be used to create a new path and return a path number, which specifies that path. The 'path' is a small data structure, which contains information about the hardware that is to be used, how that hardware has been initialised, and its current status. Each of the devices in a Minos system is given a name, which is actually the name of the module that holds the default configuration data. These names are used to identify the device for `_Open` and `_Create` system functions. For example a serial port may be called 'S0', a LCD display is called 'LCD' etc.

4.2. Device Name Syntax

The hardware independent nature of the Minos I/O routines means that they will work equally well on any device providing that a path has been opened for that device. Device names start with a ':' followed by the name of the configuration module for that device. For example, ':LCD' would be a valid device name for a LCD display. If a device is capable of supporting more than one path (i.e. a disk drive) the individual files can be identified by including sub directories and file names separating them with a '\', eg.

`':D1\C:\module\file.txt'`

This would be the name used to open a path to a file called file.txt, in a directory called module, on the 'C' drive of the host P.C., using the system's 'D1' device. In order to allow files to be accessed in the current directory of a disk device the ':' can be omitted for files only. In this case the name is assumed to be the name of a file in the current directory and the above name could be rewritten as file.txt.

4.3. Random Access Files

The hardware independent structure of the I/O system allows it to be used with devices that contain more than one file. Each file on the device can have a path opened for it and can be used in the same

way as the character based devices described above. Minos provides a number of extra functions that can be used by these multiple file devices to maintain a complete filing system. These include creating and deleting files, setting and reading the position of the next read or write operation within the file and finding the size of a file.

4.4. Configuring Devices

Each of the I/O devices in a Minos system uses two modules. One is called the driver module and contains the machine code required to drive the hardware. The other is called the device initialisation table and contains the default configuration (baud rate, parity etc.) for the device. Each time the system is reset the configuration information is copied into RAM from the initialisation table so that it can be modified before it is used to initialise the device. The configuration information is only used once when the device is initialised. This means that a device must be initialised again before any new configuration parameters will take effect. This can be done with the `_Init` function.

4.5. The Configuration Tables

Each device has a configuration table in RAM. This table has a number of fixed fields, which allows the devices to be configured without any knowledge of the hardware being used. Each field is at a fixed offset from the start of the configuration table, which can be determined, using the `_SetUp` function. The configuration fields can then be manipulated as follows:

Note:

Fields 0 to 11 are used by the system and should not be modified.

Baud Rate, Offset 12

This byte allows the more common baud rates to be set without having to know anything about the type of UART being used. To set the baud rate write one of the following codes to offset 12 of the configuration table:

Minos

0	50	1	75	2	110
3	150	4	300	5	600
6	1200	7	1800	8	2400
9	3600	10	4800	11	9600
12	19200	13	38400	14	57600

Word Format, Offset 13

This byte allows the parity, the number of data bits and the number of stop bits to be configured. The byte is divided into three fields as follows.

Note:
Bits 6 & 7 are not used.

Parity Control

	Bit 1	Bit 0	
0	0	0	No Parity
0	1	1	Odd Parity
1	0	0	Even Parity

Word Length

	Bit 3	Bit 2	
0	0	0	8 bit data
0	1	1	7 bit data
1	0	0	6 bit data
1	1	1	5 bit data

Stop Bits

	Bit 5	Bit 4	
0	0	0	1 stop bit
0	1	1	1.5 stop bits
1	0	0	2 stop bits

Number Of Columns, Offset 14

This is a two byte field for the number of columns on a display device. The LCD driver uses this field to configure the number of rows on the LCD. It is also used by keypad drivers to describe the columns in a key matrix.

Number Of Rows, Offset 16

This is a two byte field for the number of rows on a display device. The LCD driver uses this field to configure the number of rows on the LCD. It is also used by keypad drivers to describe the rows in a key matrix.

Network Station, Offset 21

This is a byte field used for the network station number. This station number should be changed in all the stations on the network before the network is used.

Examples

The first example shows how to configure a serial port called 'S0' for 300 baud, 8 data bits, one stop bit and even parity. The second shows how to use a 4 line by 40 character display.

Example 1

```
      lea      Name(pc), a0
      Minos   _SetUp
      move.b  #$04, 12(a0)
      move.b  #$03, 13(a0)
      lea      Name(pc), a0
      Minos   _Reinit
Name   dc.b   ":S0", 0
```

Example 2

```
      lea      Name(pc), a0
      Minos   _SetUp
      move.w  #40, 14(a0)
```

```
        move.w    #4,16(a0)
        lea     Name(pc),a0
        Minos   _Reinit
Name    dc.b     ":LCD",0
```

5. Writing Device Drivers

5.1. Introduction

The Minos operating system provides a complete interface between hardware devices such as serial ports, LCDs, keyboards etc. and high level application programs. This allows applications programs to be written without having to include low level support for I/O devices making them easier to write and easier to transport to other I/O devices. The Minos I/O system uses two types of module to provide the hardware interfaces. These are the device driver which contains the machine code to drive the hardware, and a device initialisation table or DIT which contains data on how each piece of hardware has to be initialised (baud rate, parity etc.) and which driver is to be used. Every device on a system has a DIT associated with it, but there needs to be only one driver for each different type of hardware. Applications programs use the name of the DIT module to identify each device.

5.2. Internal Structure of a DIT

A device initialisation table is a fixed data structure in Minos module format, which contains a number of initialisation constants for the device they refer to. During system initialisation these tables are copied into an area of RAM called the device configuration table. The configuration table is part of a larger area of RAM called the device variable space or DVS. This area of RAM is where each device keeps all of its buffer pointers and initialisation information and is passed to the appropriate driver when I/O functions are performed on that device. A DIT is a Minos module of type 1 containing a number of constants. The example below shows the size and order of the

constants. A complete example that can be used to initialise a device using a driver called 'drtest' is given later. Details of the Minos module header format in the Minos Module Format section 2.7.

```
*****
*      This is the data used by the drivers to
*      initialise each piece of hardware. DITs
*      always have this format no matter what
*      type of hardware is being used. Entries
*      which are not relevant to a particular
*      device are normally set to 0

_start      dc.l      port
            dc.l      name-_head
            dc.w      flags
            dc.w      vector
            dc.b      baud
            dc.b      parity
            dc.w      column
            dc.w      row
            dc.b      tracks
            dc.b      sectors
            dc.b      sides
            dc.b      station
name        dc.b      drtest,0
```

Port

This is the base address of the piece of hardware that this device is to use. In systems that have more than one of the same type of I/O chip the driver uses this field to determine exactly which chip is to be used when an I/O function is performed on this device.

Name

This field contains the offset from the start of the DIT to the name of the driver that is to be used when accessing this device.

Flags

This is a bit field which is used to enable and disable a number of the processing functions of the driver.

Bit 0 Character or Block based device

If this bit is clear the device is expected to perform its operations one character at a time like a serial port. If this bit is set the device moves data around in blocks like a disc.

Bit 1 Character Echo during line input

If this bit is set the Minos `_ReadLn` function will echo incoming characters to the output of the device being used for input. If this bit is clear there will be no echoing at all. Devices that do not have any output capabilities should leave this bit clear.

Bit 2 Line Repeat

If this bit is set the CTRL-A character will repeat the whole of the last line up to the first return character. This bit should only be set on devices which have input and output capabilities.

Bit 3 Add line feed

If this bit is set an extra line feed will be written to the output device after the return character at the end of a `ReadLn` function. This has no effect if bit 1 is clear.

Bit 4 Soft handshakes

This bit is reserved and should be set to 0.

The other bits are currently undefined and have no effect but they may be used in future releases.

Vector

This field contains the vector number to be used for any interrupts that may be required by the hardware.

Baud

This field contains the default baud rate code for a serial port. The codes are defines in the Configuring Devices section.

Parity

This field contains the default parity and word length code for a serial port. The codes are defined in the Configuring Devices section.

Column

This field contains the number of columns available on a display and is usually used by the LCD and graphics drivers. It can also contain the column matrix when used with keypad drivers.

Row

This field contains the number of lines available on a display and is usually used by the LCD and graphics drivers. It can also contain the row matrix when used with keypad drivers.

Tracks

This field contains the number of tracks available on a block based device like a disk drive (currently not used).

Sectors

This field contains the number of sectors per track on a block based device (currently not used).

Sides

This field contains the number of sides on a disk drive (currently not used).

Station

This field contains the network station number (currently not used).

Although the drivers supplied with Module systems all conform to this layout, the only fields that are essential to the system are the port address, the flags and the vector number. The other fields can have any meaning attached to them provided that the device driver routines use the fields accordingly.

5.3. Creating a New DIT

The example below shows the complete source code (including the Minos module header) of a DIT, which will initialise a port using 'drtest' to 9600 baud, 8 data, 1 stop and no parity. When creating a new DIT it is best to start with a piece of source code like this then change the equates at the bottom to suit the new device. When the source is complete it should be assembled to produce a binary output file in the current directory. Once the checksum, name and size of the module have been added an initialisation table called TEST is created.

```
*****
*      This is the header section which the system
*      uses to identify the module during
*      initialisation. Some of the fields will be
*      filled in by the sum program

__cstart    dc.w    $4afc          Module start marker
            dc.w    0              Space for checksum
            dc.l    0              Space for module size
            dc.l    _name-__cstart  Offset to
            module name
            dc.l    _start-__cstart Offset to code
            start
            dc.w    _type          Program type
            dc.w    0
            dc.l    0
            dc.l    0
            dc.l    0
            dc.w    $a55a          Module header end
            marker

*****
*      This is the data used by the drivers to
*      initialise each piece of hardware. DITs
*      always have this format no matter what
*      type of hardware is being used. Entries which
*      are not relevant to a particular device are
*      normally set to 0
```

```

_type      equ    1
_start     dc.l   port
           dc.l   drname-__cstart
           dc.w   flags
           dc.w   vector
           dc.b   baud
           dc.b   parity
           dc.w   column
           dc.w   row
           dc.b   tracks
           dc.b   sectors
           dc.b   sides
           dc.b   station
drname     dc.b   'drtest',0,0
_name      dc.w   0      Space for module name text

port       equ    $120100
flags      equ    $06
vector     equ    $4c
baud       equ    $0B      9600
parity     equ    $00      none
column     equ    $00
row        equ    $00
tracks     equ    $00
sectors    equ    $00
sides      equ    $00
station    equ    $00

```

5.4. Device Drivers

A device driver contains a set of five subroutines that actually manipulate the hardware devices and perform the I/O functions. A single driver can be responsible for a number of I/O devices. For example dr8530, the driver for the 85C30 UART is used for all 8 ports on an 8 channel serial expansion board. The driver is allocated a block of RAM to use for variable storage (the DVS). The operating system maintains a separate copy of this RAM for each device on the system and passes a pointer to the appropriate DVS to the driver each time an I/O operation is performed. This allows the same driver object code to be used for different devices.

5.4.1. Internal Structure of a DVS Block

Each DVS block is 64 bytes long and is created during system startup by the operating system each time a DIT module is located. A copy is made in the configuration table area of the DVS of all the setup information in the DIT. The DVS also contains space for all the pointers required to maintain FIFO buffers for the receiver and transmitter. The last 14 bytes of the DVS are not used by the operating system and can be used for any extra data required by special hardware.

Offset	Name	Size
\$00	DVS_Rxb	4
\$04	DVS_Txb	4
\$08	DVS_Esc	4
\$0C	DVS_Rxi	2
\$0E	DVS_Rxr	2
\$10	DVS_Txi	2
\$12	DVS_Txr	2
\$14	DVS_Rdy	2
\$16	DVS_Flg	2
\$18	DVS_Pid	2
\$1A	DVS_Sig	2
\$1C	DVS_Port	4

\$20	DVS_Drvr	4
\$24	DVS_Type	2
\$26	DVS_Irq	2
\$28	DVS_Baud	1
\$29	DVS_Parity	1
\$2A	DVS_X	2
\$2C	DVS_Y	2
\$2E	DVS_Tks	1
\$2F	DVS_Sec	1
\$30	DVS_Head	1
\$31	DVS_Net	1
\$32	DVS_User	14

_DVS_Rxb Pointer to the RAM used for the receiver buffer

This is a 32 bit number and is used to hold the address of the RAM which is being used for the receiver FIFO buffer. This RAM should be claimed using Minos `_Malloc` when the device is first initialised.

_DVS_Txb Pointer to the RAM used for the transmitter buffer

This is a 32 bit number and is used to hold the address of the RAM which is being used for the transmitter FIFO buffer. This RAM should be claimed when the device is first initialised.

_DVS_Esc Pointer to the Escape event flag

This is a 32 bit number which contains the address of the location which the application program is using for its escape flag.

_DVS_Rxi Receiver Buffer Insert Index

This is a 16 bit number and is used as the index into the receiver FIFO where the next byte received by the receiver interrupt service routine should be placed.

_DVS_Rxr Receiver Buffer Remove Index

This is a 16 bit number and is used as the index into the receiver buffer where the next character requested by `_Read` or `_ReadLn` should be taken from.

_DVS_Txi Transmitter Buffer Insert Index

This is a 16 bit number and is used as the index into the transmitter buffer where the next character sent by `_Write` should be placed.

_DVS_Txr Transmitter Buffer Remove Index

This is a 16 bit number and is used as the index into the transmitter buffer where the next character is to be taken from by the transmitter interrupt service routine.

_DVS_Rdy Number of Characters Ready

This is a 16 bit number and is used to hold the number of characters which are waiting in the receiver FIFO to be picked up by `_Read` or `_ReadLn`.

_DVS_Flg Special System Flags

This is a 16 bit flag field which is used by the operating system to keep track of how the device is being used.

_DVS_Pid Waiting Process ID

This is a 16 bit field which contains the process ID of the process which is currently waiting to send some output on this device.

_DVS_Sig Signal Code

This is a 16 bit field which contains the process ID of the process which is currently waiting for data to become available.

_DVS_Port Port Address

This is a 32 bit number and is a copy of the port address field from the DIT.

_DVS_Drvr Driver Start Address

This is a 32 bit number which contains the execution entry address of the driver for the device referred to by this DVS block. This address is calculated by the operating system during startup.

The remaining fields in the DVS block contain the current configuration information for the device and are collectively known as the device configuration table.

_DVS_Type Device Flags

This is a 16 bit number which contains a copy of the device flags from the DIT.

_DVS_Irq Interrupt Vector Number

This is a 16 bit number which contains a copy of the vector number in the DIT.

_DVS_Baud Baud Rate Code

This is an 8 bit number which contains a copy of the baud rate code in the DIT.

_DVS_Parity Parity Code

This is an 8 bit number which contains a copy of the parity and word format control code in the DIT.

_DVS_X Number of Columns

This is a 16 bit number which contains a copy of the number of columns on a display in the DIT.

_DVS_Y Number of Rows

This is a 16 bit number which contains a copy of the number of rows on a display in the DIT.

_DVS_Tks Number of Tracks

This is an 8 bit number which contains a copy of the tracks field in the DIT.

_DVS_Sec Number of Sectors

This is an 8 bit field which contains a copy of the sectors field in the DIT.

_DVS_Head Number of Heads

This is an 8 bit number which contains a copy of the heads field in the DIT.

_DVS_Net Station Number

This is an 8 bit number which contains a copy of the station field in the DIT.

_DVS_User User Area

This area is 14 bytes long and is not used by the system in any way. It can be used in drivers to hold useful data related to specific hardware like RAM copies of write only registers etc.

5.4.2. Internal Structure of a Device Driver

The device driver is a set of routines contained in a Minos module which actually work the hardware. The format of a driver module is similar to 68000 program modules except that the execution entry must point at a number of long branches to the required routines as follows:

```
*****
*      The first section is a standard Minos module
*      header with a module type of 2 (device
*      driver)
```

```
_head      dc.w      $4AFC
           dc.w      0
           dc.l      0
           dc.l      0
           dc.l      _start-_head
           dc.w      _type
           dc.l      0
           dc.l      0
           dc.l      0
           dc.w      0
           dc.w      $A55A
```

```
*****
*      The code at the execution entry point must be
*      a set of five long branches to the driver
*      routines in the correct order
```

```
_type      equ      2
_start     bra      init
           bra      read
           bra      write
           bra      functions
           bra      deinit
```

The rest of this section describes the functions that must be provided by each routine and which registers contain parameters etc.

5.4.2.1. The Initialisation Routine

The initialisation routine, as its name implies, is responsible for setting up the hardware and preparing the device for use. This routine is called when the first `_Open` or `_Create` call is made to a device. This means that devices will only take up buffer and IRQ table space if they are actually needed and also that there is no chance of making I/O function requests on uninitialised hardware. The 'init' routine is also called explicitly by the Minos `_ReInit` function, which allows the hardware set up of a device to be modified while the system is running.

Register Usage During Init

When 'init' is called the following registers will have already been set up:

a2	The address of the DVS block for this device
a3	The base address for the hardware

During the initialisation routine registers a0, a4, d0, d2 and the condition code register can be freely used without preserving them. All the other registers must be preserved. When the initialisation is complete it should return with the carry bit clear if the initialisation has worked correctly or with the carry bit set and an error code in d1.I. The error codes that can be returned by the driver are defined in the Minos Error Codes section 7.

The code in the initialisation routine is very hardware dependent but it is usually best to follow the following general flow. Start by claiming

enough memory for any buffers required using Minos `_Malloc` and put the addresses of the buffers into the DVS block at `_DVS_Rxb` and `_DVS_Txb`. If an error occurs in the memory claim, return immediately with carry set, otherwise clear all the indexes and the number of characters received. If the device is to be interrupt driven install an interrupt service routine on the vector given in `_DVS_Irq` using Minos `_Irq`. If this fails return the buffer memory using Minos `_Dalloc` and return with carry set. Now initialise the hardware with the appropriate parameters from `_DVS_Baud`, `_DVS_Parity` etc. Finally enable the interrupts from the hardware (if there are any) and return with carry clear. The example below shows a very simple initialisation routine for the MC68681 serial port that does not use interrupts. In order to create the complete driver module, assemble the source into a binary file called `drtest` in the current directory and complete the header. This will give the driver a module name of `drtest` so that it can be used directly with the DIT module developed earlier.

```
*****
*   Initialise the 68681 serial port. This driver
*   does not use any interrupts so it does not
*   claim buffer memory or install an interrupt
*   service routine.
*       a2 - Pointer to DVS
*       a3 - Hardware base address
*   Can destroy a0 a4 and d0 d2
*   return with C = 0 or C = 1 and d1.l = error
*   number

Umr   equ       $01
Usr   equ       $03
Ucs   equ       $05
Ucr   equ       $07
Txd   equ       $09
Rxd   equ       $0A

init  clr.w      d0
      move.b     _DVS_Baud(a2),d0  Baud rate code
      lea       Baud_Tab(pc),a0   Baud table
```

```

    move.b    0(a0,d0.w),Ucs(a3)Set baud rate
    move.b    #$31,d1
    move.b    _DVS_Parity(a2),d0Read default
    asr.b     #$01,d0
    bcc.s     No_Par           No parity
    bset      #$03,d1         Parity is on
    asr.b     #$01,d0
    bcc.s     No_Par           Odd parity
                                needed
    bset      #$02,d1         Parity is even

No_Par
    move.b    _DVS_Parity(a2),d0
    asr.b     #$03,d0
    bcc.s     d8              8 bit chars
    bclr      #$00,d1         7 bit chars
d8
    asr.b     #$03,d0
    bcc.s     Stops1
    bset      #$01,d1

Stops1
    move.b    d1,Umr(a3)
    move.b    #$05,Ucr(a3)    Enable Rx & Tx
    rts

Baud_Tab
    dc.b     $08
    dc.b     $08
    dc.b     $38
    dc.b     $19
    dc.b     $2A
    dc.b     $03
    dc.b     $3B
    dc.b     $6E
    dc.b     $4C
    dc.b     $6E
    dc.b     $5D
    dc.b     $6E
    dc.b     $7F

```

dc.b \$7F

5.4.2.2. The Read Routine

This routine provides the interface between the operating systems `_Read` and `_ReadLn` functions and the hardware. The operating system always reads from its devices in blocks which minimises I/O overheads. The driver read routine should take the requested number of characters either from its buffer or directly from the hardware and place them into the application programs buffer. If the input buffer does not contain enough characters to meet the request the read routine should wait for more to come in.

Register Usage During Read

When read is called the following registers will have already been set up:

d1	The number of bytes to read
a0	Pointer to the application programs buffer
a2	Pointer to the DVS block for this device
a3	Base address of the hardware

During the read routine registers a1, a4, d0, d2 and the condition code register may be freely used. When the read is complete it should return with C = 0, a0 still pointing at the application buffer and d1 containing the number of characters actually read (usually the number requested). If the read fails for some reason it should return with C = 1 and an error code in d1.l. The routine below shows a simple polled read routine for use with an MC68681.

```
*****
*   Poll the 68681 UART receiver to read a block
*   of data.
*   d1.w - Number of bytes to read
*   a0 - Address of application programs buffer
*   a2 - Address of DVS block for this device
*   a3 - Hardware base address
*   May destroy a1, a4, d0, d2 and the condition
*   codes.
*   Returns
```

```

*           C = 0,      a0 Preserved
*           dl.w number of bytes read
*           C = 1,      a0 Preserved
*           dl.l error number

read  movem.l    d1/a0,(a7)  Save some registers
      subq.w     #$01,d1     For dbra instruction
loopr btst      #$00,Usr(a3)Character ready
      beq.s     loopr      Round again if not
      move.b    Rxb(a3),(a0)+ read character
      dbra     d1,loopr    Loop for all chars
      movem.l   (a7)+,d1/a0 Recover registers
      rts

```

5.4.2.3. The Write Routine

This routine provides the interface between the operating systems `_Write` function and the hardware. The operating system always writes to its devices in blocks which minimises I/O overheads. The driver write routine should send the requested number of characters from the application programs buffer and write them either to its buffer or directly to the hardware. If the output buffer fills up during the write request the write routine must wait for more space to become available to ensure that all the characters are sent.

Register Usage During Write

When write is called the following registers will have already been set up:

d1	The number of bytes to be written
a0	Pointer to the application programs buffer
a2	Pointer to the DVS block for this device
a3	Base address of the hardware

During the write routine registers a1, a4, d0, d2 and the condition code register may be freely used. When the write is complete it should return with C = 0, a0 still pointing at the application buffer and d1 containing the number of characters actually written (usually the number requested). If the write fails for some reason it should return

with C = 1 and an error code in d1.l. The routine below shows a simple polled write routine for use with an MC68681.

```
*****
*   Poll the 68681 UART transmitter to write a
*   block of data.
*   d1.w - Number of bytes to write
*   a0 - Address of application programs buffer
*   a2 - Address of DVS block for this device
*   a3 - Hardware base address
*   May destroy a1, a4, d0, d2 and the condition
*   codes
*   Returns
*   C = 0      a0 Preserved
*              d1.w number of bytes written
*   C = 1      a0 Preserved
*              d1.l error number

write movem.l    d1/a0,(a7)  Save some registers
      subq.w     #$01,d1     For dbra instruction
loopw btst      #$02,Usr(a3)Transmitter ready
      beq.s     loopw       Round again if not
      move.b    (a0)+,Txd(a3) Put char into
                              transmitter
      dbra     d1,loopw     Loop for all chars
      movem.l  (a7)+,d1/a0 Recover registers
      rts
```

5.4.2.4. The Special Functions (_Functions)

The special function routines are used to provide all the extra little bits that do not fit neatly into the other routines. The CMS drivers use a number of special functions for the serial file server etc. but there are only five that need to be provided to make a driver work correctly.

Code 0, which returns the number of characters ready.

Code 1, which returns the number of empty spaces in the transmitter buffer.

Code 16, which is called when a path is opened.
Code 17, which is called when a path is closed.
Code 18, which is called when a path is created.

If a driver does not support a function it should return with carry set and error code 10 (`_E$Func`) in `d1.l`.

Register Usage during Special Functions

When a special function is requested by the operating system the following registers will have already been set up:

`d1.w` The code number requested
`a2` The address of the DVS block for this device
`a3` The base address of the hardware
Other registers depend on the code being used.

During a special function routine the following registers may be used freely `a0`, `a4`, `d0`, `d2` and the condition code register. Other registers may be used depending on the function being called. If the function is successful it should return with carry clear otherwise it should return with carry set and an error code in `d1.l`. The example below shows a minimal special function service routine for use with an MC68681.

```
*****  
* Special function services for the 68681  
* polled driver  
* d1.w - function number  
* a2 - Address of DVS block for this device  
* a3 - Base address of the hardware  
* Returns  
* C = 0 registers depend on function  
* C = 1 d1.l error code
```

```
functions  
    cmpi.w    #$00,d1    Chars ready call  
    bne.s     Not_Rdy  
    clr.l     d0         Assume no chars ready  
    btst     #$02,Usr(a3)Check receiver status  
    beq.s     No_Char
```

```
        move.l    #$01,d0
No_Char
        rts
Not_Rdy
        cmpi.w   #$10,d1    Open path call
        bne.s   Not_Opn
        clr.l   d1         Nothing to do set C=0
        rts
Not_Opn
        cmpi.w   #$11,d1    Close path call
        bne.s   Not_Clo
        clr.l   d1         Nothing to do set C=0
        rts
Not_Clo
        move.l   #10,d1     _E$Func error code
        ori.b   #01,CCR    Set carry bit
        rts
```

5.4.2.5. The Deinitialisation Routine

This is the last of the required routines and is responsible for closing down a device neatly so that it can be removed or reinitialised with different parameters. The deinitialisation routine is basically the reverse of the initialisation routine and should approximately follow this pattern. Firstly wait for the transmitter buffer to be emptied if the transmitter is being interrupt driven. Shut down the hardware and make sure that it stops generating interrupts. Remove the interrupt service routine from the system using Minos _Irq and finally return any memory claimed by the initialisation routine using Minos _Dalloc.

Register Usage During Deinit

When deinit is called the following registers will have already been set up:

- a2 The address of the DVS block for this device
- a3 The base address for the hardware

During the deinitialisation routine registers a0, a4, d0, d2 and the condition code register can be freely used without preserving them, all the other registers must be preserved. When the deinitialisation is

complete it should return with the carry bit clear if the deinitialisation has worked correctly or with the carry bit set and an error code in d1.l.

```
*****
*   Deinitialise the 68681 serial port. This
*   driver does not use any interrupts so there
*   is very little to do here, just wait for the
*   last character to be sent then disable the
*   line drivers.
*   a2 - Pointer to DVS
*   a3 - Hardware base address
*   Can destroy a0, a4 and d0, d2
*   Return with
*   C = 0 or C = 1 and d1.l = error number
```

```
deinit
    btst      #$02,Usr(a3)      Tx empty
    beq.s    deinit
    move.b   #$3F,Apor        Disable line
                                drivers
    rts
```

5.5. Testing New Drivers

New driver and DIT modules can be loaded at any time using the normal load command in the same way as programs are loaded. The only thing to watch out for is that the driver must be loaded before the DIT. To load the driver and DIT from the above examples type:

```
load drtest      <return>
load TEST        <return>
```

The new driver can now be used in the same way as the ones already in the PROM, e.g.

```
fp = fopen(:TEST,"r+");
fprintf(fp,"Hello from the new port\n");
```

If the new driver does not work for some reason it can be debugged using the on board machine code debugger as follows. Firstly reset the Module and load the driver and DIT as before. Locate the address of the driver using the 'mdir' utility. Enter the debugger by typing 'd'. Use the debugger to disassemble the branch table, which is 34 bytes on from the start of the driver. Set a breakpoint in the initialisation routine in the driver. Return to the shell by typing 'g' at the debugger command line. Send some output to the new driver (which will call its initialisation routine). The breakpoint will be reached and the system will enter the debugger with all the registers etc. ready to perform the drivers initialisation routine. The routine can now be traced by the debugger, registers can be modified etc. When the initialisation has been completed successfully another breakpoint can be set in the write routine so that it can be debugged in a similar way. Finally type 'g' to allow the system to finish opening the path and return to the command line.

6. Minos System Functions

6.1. Introduction

This table gives a list of all of the functions available along with their function number and a recommended label.

Parameters

All parameters are long.

Strings must be terminated with \$00.

All registers are preserved except for those specified.

Status register returns are shown by capitals

Lower case is unspecified

Function	Function No	Description
_Malloc	0	Allocate a block of memory
_Dalloc	1	Return a block of memory
_Fix_Mod	2	Add a module to the system
_Link	3	Find an existing module
_UnLink	4	Remove an existing module
_Fork	5	Create a process from a subroutine
_Chain	6	Create a process from a module
_Exit	7	Terminate a process
_Sleep	8	Make a process idle for a time
_Wait	9	Make a process permanently idle
_Signal	10	Send a signal to a process
_ReadSys	11	Read a system state register
_Irq	12	Fit routine into IRQ polling table
_Open	13	Create a path to a file or device
_Read	14	Read a block of binary data
_ReadLn	15	Read a line of characters
_Write	16	Write a block of binary data
_WriteSys	17	Write to system state registers
_Function	18	Perform special device functions
_Close	19	Close down an I/O path

_WrTick	20	Write to elapsed time counter
_RdTick	21	Read from elapsed time counter
_UsrRam	22	Return start of safe RAM area
_Escape	23	Turn the escape key on or off
_Ready	24	Return number of characters ready
_Dog	25	Change state of watchdog trigger
_Create	26	Create a file by name
_Delete	27	Delete a file by name
_SetPos	28	Set file pointer position
_GetPos	29	Get file pointer position
_Setup	30	Get address of configuration table
_Relnit	31	Force a device to reinitialise
_ModDir	32	Get start of system module list
_Procs	33	Get pointer to process table
_UnFix	34	Remove a module from the list
_Fsize	35	Return the length of a file
_BackUp	36	Battery back this memory
_UnBack	37	Remove this backed up memory
_Trim	38	Trim down this memory
_Debug	39	Enter the 68000 debug monitor
_Vectors	40	Start of 68000 vector table
_Di	41	Disable interrupts & process swaps
_Ei	42	Enable interrupts
_ResTyp	43	Read last reset type
_MkData	44	Create a data module
_Death	45	Send signal as process dies
_Echo	46	Change echo device for line input
_Getime	47	Read from the Real Time clock
_Setime	48	Set the Real Time clock
_Delnit	49	Call a drivers 'deinit' routine

6.2. **_BackUp**

Function Code 36

Entry:

a0 Address of the previously allocated memory block to
 be backed

Exit:

xnzvC
C=0 OK d1.l undefined
C=1 error d1.l Error code

This function is used to preserve a block of type 0 memory during power failure. The address given should be the one returned by `_Malloc` when the block was claimed. If the requested area is not in type 0 an error will be returned.

6.3. **_Chain**

Function code 6

Entry:

a0 The address of the name of the program module
 terminated with \$00.
d0.l The path number to use for stdin & stdout
d1.l The stack size for the new process

Exit:

xnzvC
C=0 OK d1.l undefined
C=1 error d1.l error code

This function creates a new process. A link call is made to locate the program module for the new process. If it is found, a block of memory is claimed for the process descriptor and stack for the new process. The size of the stack for the new process is in d1. The minimum size is 1024 bytes and smaller stack sizes will be rounded up. The data registers d1 - d7 and the address registers a1 - a6 are saved so that the new process will inherit them.

Note:

The initial value of d1 will be the actual stack size for this process, which will not necessarily be the size that was requested. Register d0 contains the path number to be used as the standard input and output path (path 0) for the new process. If d0 is -1, the new process will be given the same standard input and output path as the process which created it.

6.4. _Close

Function Code 19

Entry:

d0.l Path number

Exit:

xnzvC

C=0 OK d1.l undefined

C=1 error d1.l Error code

This function is used to close a path when it is no longer required. It is not really necessary to close down paths to devices however it is important to close paths to files to ensure that any sector buffers are written onto the disk. Paths that are left open will use up memory and it is good practice to close paths if they are not going to be used again.

6.5. _Create

Function Code 26

Entry:

a0 Points to the name of the file in ASCII terminated in \$00

Exit:

xnzvC

C=0 OK d0.l Path number for the new file

d1.l undefined

C=1 error d1.l Error code

This function is used to create a new file and returns a path number for it. It performs in a similar way to _Open except that a new file is created. If the file already exists the old file is deleted before the new one is created. If _Create is used on a device it has exactly the same effect as _Open.

6.6. **_Dalloc**

Function Code 1

Entry:

a0 Address of the RAM to be de-allocated

Exit:

xnzc

C=0 OK d1.l undefined

C=1 error d1.l Error code

This function is used to return memory that was previously claimed using `_Malloc`. The address given should be the one returned by `_Malloc` when the claim was made otherwise an error will be returned.

Note:

It is not possible to return a block of memory that is marked as battery backed.

6.7. **_Death**

Function code 45

Entry:

d0 - Process ID of the process which must send the signal

Exit:

xnzc

C=0 OK d1.l undefined

C=1 error d1.l Error code

This function is used to request a signal to be sent by the process whose process ID number is passed in d0 when it terminates. The signal is sent to the process that calls `_Death`. This function allows processes to be run sequentially.

6.8. `_Debug`

Function code 39

Entry:

all registers passed to the debug monitor

Exit:

If the registers are not changed, the debug command 'g' will return after the trap.

This drops into the 68000 machine code debugger. All the current registers are recorded and can be displayed using the debug 'r' command. All the normal debug commands will work including trace. A 'g' will return the program directly after the trap call if the PC and stack registers are intact. The data and address registers may be changed and their new values will be passed into the program.

6.9. `_Delete`

Function code 27

Entry:

a0 Points to the name of the file in ASCII terminated with \$00.

Exit:

xnzvC			
C=0	OK	d1.l	undefined
C=1	error	d1.l	Error code

This function deletes a file by name. The file must be closed before it can be deleted. If delete is called with a non-existent file name it returns without an error. If it is called on a device that cannot support files an error is returned.

6.10. **_Di**

Function Code 41

Entry:

None

Exit:

xnzvc

d1.l undefined

Disable interrupts and stop task swapping.

6.11. **_Dog**

Function Code 25

Entry:

d0.l	0	Turn off dog light
	+ve	Turn on dog light
	-ve	Invert dog light

Exit:

xnzvc

d1.l undefined

This function is used to drive the watchdog. If the watchdog is enabled this function must be called every 360 ms with a -ve value in d0 to prevent the watchdog from timing out. The watch dog timer is 400 ms \pm 10%.

6.12. `_Echo`

Function Code 46

Entry:

d0.l - Path being used for input

d1.l - Path to be used for echoed characters

Exit:

xnzvC

C=0 OK d1.l undefined

C=1 error d1.l Error code

This function allows the device which '`_ReadLn`' uses to echo the input characters to be changed. D0 should contain the path that '`_ReadLn`' is using for input and d1 should contain a path for the device to be used for the echoed characters. This function is useful for input devices which do not have output devices associated with the like keypads etc.

6.13. `_Ei`

Function Code 42

Entry:

none

Exit:

xnzvc

d1.l undefined

Enable interrupts

6.14. `_Escape`

Function Code 23

Entry:

a0	Address of the escape word
d0.l	0 disable escape effect
	<> 0 enable escape effect

Exit:

xnzvc	
d1.l	undefined

This function is used to control the action of the 'ESC' key on the terminal used by path zero. If d0 is zero then the escape action is disabled but if it is non zero the 16 bit number at the address in a0 will be set to a non zero value whenever the 'ESC' key is pressed. The application is responsible for clearing this flag after taking the appropriate action.

6.15. `_Exit`

Function code 7

Entry:

None

Exit:

This function does not return

`_Exit` is used to terminate a process. When it is called the system removes the process from all the queues, de-allocates the memory used by the process descriptor and moves on to the next process. Each process should make sure that it closes any paths that it created and returns any memory that it claimed before calling `_Exit`.

6.16. `_Fix_Mod`

Function code 2

Entry:

a0 The address of the start of a module (the \$4AFC)

Exit:

xnzc

C=0 OK d1.l undefined

C=1 error d1.l Error code

This function allows new modules to be attached to the system while it is running. It starts by checking that the module header start and end markers are correct, then checks the checksum. If all these are correct then the module is added to the module directory. If the module is a device initialisation table, the configuration data is also copied into RAM.

6.17. `_Fsize`

Function code 35

Entry:

d0.l The path number of the file

Exit:

xnzc

C=0 OK d0.l The size of the file

d1.l undefined

C=1 error d1.l Error code

This function returns the size of a file. This function can be used in conjunction with `_Open` and `_SetPos` to open a file for appending, i.e. at the end of the file. Use `_Open` to return a path number for the file. Then find the length using `_Fsize` and finally set the file pointer to the file size using `_SetPos`. The next write operation will start writing at the first byte after the end of the file.

6.18. **_Functions**

Function Code 18

Entry:

d0.l Path number
d1.l Function code number

Exit:

dependent on function

The special function call provides a convenient method of adding hardware and driver dependent routines, which cannot be included in the normal read, write and initialise calls. The number in d1 identifies each function. Functions 0 <196> 255 are reserved for use by the system other user functions can be supplied with codes 256 <196> 65535. The special function call has some common parameters and some, which are specific to each particular function.

Code 0 _F_Ready

This function returns the number of characters waiting in the input buffer of the device.

Exit:

xnzvc
d1.l number of bytes in input buffer

Code 1 _F_Free

This function returns the number of empty places in the output buffer of a device. This allows programs to ensure that they will not be suspended when they use the '_Write' function.

Exit:

xnzvc
d1.l number of spaces in the output buffer

Code 2 _F_Getch

This function performs a single character read from the input device. This read is non-interrupt driven and does not use the internal buffers. It is used extensively by the debugger.

Exit:

xnzvc

d2.l input character or -1 if no characters are available

Code 3 _F_Putch

This function performs a single character write to the output device. This write is non-interrupt driven and does not use the internal buffers. It is used extensively by the debugger.

Exit:

xnzvc

d2.l input character or -1 if no characters are available

Code 16 _F_Open

This function is called whenever a new path is opened on a device. Drivers which do not need to do anything during open should simply return with the carry clear.

Exit:

xnzvc

d1.l undefined

Code 17 _F_Close

This function is called whenever a path is closed. Drivers which do not need to do anything during close should return with the carry clear.

Exit:

xnzvc

d1.l undefined

Code 18 _F_Create

This function is called whenever a path is created. Drivers which do not need to do anything during close should return with the carry clear.

Exit:

xnzvc

d1.l undefined

Code 138 Enable the filing system support.**Code 139 Disable the filing system support.**

Exit:

xnzvc

d1.l undefined

The serial port used for the filing system can have the special functions disabled so that it can be used for pure binary data.

Note:

That the Escape effect will also have to be disabled to allow binary I/O (See `_Escape`).

There are a number of other special function routines which are not documented here. These are used by the 'PC' hosted filing system and cannot be used safely by application programs.

6.19. `_GetPos`

Function code 29

Entry:

d0.l The path number of the file

Exit:

xnzvC

C=0 OK d0.l The current file pointer position

 d1.l undefined

C=1 error d1.l Error code

This function is used to return the current value of a file pointer. An error is returned if this function is used on a device which does not support files.

6.20. **_Gettime**

Function Code 47

Entry:

a0.l address of time structure

This function is used to read the real time clock. As there are so many parameters required by this function a parameter block is used rather than registers. This also makes it easier to use from high level languages. The format of the parameter block is as follows, each entry is four bytes long.

Week Day
Year
Month
Date
Hours
Minutes
Seconds

6.21. `_Irq`

Function code 12

Entry:

d0.w	CPU vector number to use for this interrupt
a0	Address of the interrupt service routine
a2	Address of the IRQ work space

Exit:

xnzvC			
C=0	OK	d1.l	undefined
C=1	error	d1.l	Error code

This routine is used to install an interrupt service routine into the system allowing the system to look after all the interrupt polling. d0.w should contain the vector number which will be taken when this interrupt occurs. A0 should contain the address of the interrupt service routine and a2 should contain the address of the service routine's workspace. In fact a2 is simply passed to the service routine unchanged so it can be used as a general 32 bit parameter for the service routine. When an interrupt service is entered a1 - a4 and d0 - d2 will already have been saved so these registers can be freely used. a2 will contain the same value as it had when the `_Irq` call was made to install the service routine. The service routine should clear the interrupt and the carry flag and return using 'rts' not 'rte'. If the routine is unable to service the interrupt it should return immediately with the carry flag set. To remove an interrupt service this function must be called with a0=0.

6.22. `_Link`

Function code 3

Entry:

a0 The address of the name of a module
d0.l The type of that module

Exit:

xnzvC
C=0 a0 Updated passed the module name
 a1 address of the modules header (\$4AFC)
 d1.l undefined
C=1 error d1.l Error code

This function searches the module directory for a module. If it finds one with the same name it checks that the type is correct then returns the start address of the module (The \$4AFC).

6.23. **_Malloc**

Function Code 0

Entry:

d0.l Memory Size
d1.l Memory Type

Exit:

xnzvC
C=0 a0 The address of the start of the RAM
 d1.l undefined
C=1 error d1.l Error code

This function is used to claim memory from the memory management system. The size of the claim is rounded up to the nearest 32 byte boundary. The memory segments are then searched in turn for a contiguous area big enough for the claim starting with the first segment of the requested type. If there is no memory of the required type or that area is full the search continues through the other type. When a block of the required size is found it is cleared to 0.

Note:

No error is returned if the allocation was successful but was not of the correct type. It is the responsibility of the application program to ensure that any memory that is claimed is eventually de-allocated.

6.24. **_MkData**

Function Code 44

Entry:

d0.l Size of the data area required in bytes
d1.l Type of memory to be used
a0.l Address of the name of the data module

Exit

xnzvC
C=0 a0 The address of the data module header
 d1.l undefined
C=1 error d1.l Error code

This function creates a data module. Data modules are simply areas of memory which have a Minos module header attached to the front so that they can be located using the '_Link' function. Data modules can be battery backed in the same way as other modules which makes them especially useful for storing program data and configuration information in battery backed memory.

6.25. **_ModDir**

Function code 32

Entry:

none

Exit:

xnzvc
a0 points to the start of the RAM table

This function returns the start of the table containing the module directory entries.

6.26. **_Open**

Function code 13

Entry:

a0 Address of the file name

Exit:

xnzvC

C=0 OK d0.l Path number

 d1.l undefined

C=1 error d1.l Error code

This function creates a path for use with an I/O device or a file. The file name should be an ASCII string that ends in 0. This call also keeps track of which devices have been initialised. If the open call finds that a device has not been initialised it will call an initialisation routine for that device and gives it an opportunity to set up interrupt services, get memory for buffers etc.

Note:

This means that devices such as serial ports cannot start to receive data until at least one path has been opened.

6.27. **_Procs**

Function code 33

Entry:

none

Exit:

xnzvc

a0 points to the start of the RAM table

This function returns the start of the table containing the current process status.

6.28. `_RdTick`

Function Code 21

Entry:

none

Exit:

d1.l Returns current value of the counter

This function is used to read the current value of the elapsed time counter.

6.29. `_Read`

Function Code 14

Entry:

d0.l Path number

d1.l The number of bytes to read

a0 Address of a buffer for the bytes.

Exit:

xnzvC

C=0 OK d1.l The number of bytes read

C=1 error d1.l Error code

This function performs raw input on a device or file. Data is read in and placed directly into the buffer. The control characters have no special effect and there is no echoing. If this function is used with a path to a character type device like a serial port it will wait until the number of characters requested have been read in. If it is made on a multiple file device and there are not enough characters in the file to satisfy the request it will read as many characters as possible and return the number actually read. If it is used on a file which is already at the end of the file it returns immediately with an end of file error. This means that files can be read in large blocks without knowing the file length.

6.30. `_ReadLn`

Function Code 15

Entry:

d0.l Path number to read from
d1.l Maximum number of characters
a0 Address of a buffer for the line

Exit:

xnzvC
C=0 OK d1.l The number of characters read
C=1 error d1.l Error code

This function is similar to `read` except that the call returns when the first carriage return (`$0D`) character is received. This function behaves slightly differently depending on whether it is used on a character device or a file. If it is used on a file it reads characters into the buffer until a `$0D` is encountered. A `$00` is inserted into the buffer instead of the `$0D` and the function returns with the number of bytes read. If buffer fills to one less than the maximum before a `$0D` is encountered the last buffer position is set to 0 and the function returns the number of bytes read. If the end of the file is reached before a `$0D` a 0 is added to the end of the buffer. If the file is already at end of file this function returns immediately with an end of file error. If this function is called on a character device all incoming characters are echoed and the control characters have the following meaning:

<code>\$01</code>	CTRL-A	Repeat the whole of the last line
<code>\$08</code>	CTRL-H	Delete one character (Backspace)
<code>\$0D</code>	CTRL-M	End of line (carriage return)
<code>\$18</code>	CTRL-X	Delete the whole line

When the first `$0D` character is encountered it is replaced by a 0 in the buffer and the function returns the number of bytes read. If the maximum number of characters -1 are read in without an `$0D` all subsequent characters are echoed as a bell (character `$07`) and no more bytes are added to the buffer. If the escape key (character 27) is read the function returns immediately with an error.

6.31. `_Ready`

Function Code 24

Entry:

d0.l Path number to check on.

Exit:

xnzvC

C=0 OK d1.l Returned number of characters ready

C=1 error d1.l Error code

This function returns the number of characters waiting in the input buffer of a device and returns the number in d1. This can be used to ensure that programs will never be held up by a `_Read` or `_ReadLn` call.

6.32. `_Reinit`

Function code 31

Entry:

a0 Points to the name of the device in ASCII terminated in \$00.

Exit:

xnzvC

C=0 OK d1.l undefined

C=1 error d1.l Error code

This function causes a device to initialise itself using the configuration table in RAM. If the device is already in use it will be deinitialised first. This means that incoming data may be lost as the interrupts will be turned off for a time. If a device is still sending data it will not be deinitialised until all the data has gone. If this function is used on a file it will initialise the serial file server.

6.33. **_ResTyp**

Function Code 43

Entry:

none

Exit:

d0.l	0	power On
	1	watch dog time out

Test reset type

Note:

It is not possible to accurately determine the type of the reset when the reset button is used. If the watchdog is enabled when the reset button is pressed the reset will show up as a watchdog reset. If the watchdog is disabled the reset type cannot be determined.

6.34. **_SetPos**

Function code 28

Entry:

d0.l	The path number of the file
d1.l	The new file pointer position

Exit:

xnzvC			
C=0	OK	d1.l	undefined
C=1	error	d1.l	Error code

This function is used to move the file pointer without writing or reading from the file. This allows new data to be read or written to anywhere in a file. If the file pointer is moved passed the end of a file the `_Read` and `_ReadLn` functions will return an end of file error but the `_Write` function will expand the file and write data at the new position. An error is returned if this function is used on a device which does not support files.

6.35. **_Setime**

Function Code 48

Entry:

a0.l - address of time structure

This function is used to set the real time clock. As there are so many parameters required by this function a parameter block is used rather than registers. This also makes it easier to use from high level languages. The format of the parameter block is as follows, each entry is four bytes long.

Week Day
Year
Month
Date
Hours
Minutes
Seconds

6.36. **_Setup**

Function code 30

Entry:

a0.l The address of the device name

Exit:

xnzvC

C=0 a0 The address of the configuration table

 a1 The address of the default settings

 d1.l undefined

C=1 error d1.l Error code

This function is used to find the configuration table for a device. The function searches the list of available devices and returns the RAM copy of the table, which is the one used to configure the device, in a0 and the PROM copy in a1. The configuration of the device can then be modified using the offsets described in the Device Configuration section. If this function is used with a file name it will return the configuration table for the file server.

Note:

The new configuration will not take effect until the `_Reinit` function is used.

6.37. **_Signal**

Function 10

Entry:

d0.l The process id of the recipient
d1.l Message to send

Exit:

xnzvC
C=0 OK d1.l undefined
C=1 error d1.l Error code

This function signals a waiting process to restart. It also sends a message that can be used by the waiting process. This causes the process to be moved from the waiting queue into the running queue so that it will run when its time slot comes around.

6.38. `_Sleep`

Function code 8

Entry:

d0.l The length of delay in 1/100 second

Exit:

xnzvC

C=0 OK d1.l undefined

C=1 error d1.l Error code

This function is used to create accurate time delays and returns to the application program after a fixed delay. The delay counter is driven under interrupt at 100 Hz by a hardware counter. There is a skew delay of up to 10 ms on each delay as there is no way of knowing how far the hardware counter is through the current tick when `_Sleep` is called. When the delay expires the sleeping process is inserted at the top of the running queue so that it becomes the next process to run. It then remains in the running queue following the normal round robin schedule. When a process is put to sleep any time remaining in the current 10 ms time slot is added to the slot for the next process. This ensures that when a process is scheduled it gets at least 10 ms. A delay of 0 is a special case and simply causes a task swap without delaying the calling process. If a process is sent a signal while it is asleep it is woken up immediately and becomes the next process to run. After being woken by a signal `_Sleep` returns an error in d1 and the number of ticks remaining in the delay in d0.

6.39. `_Trim`

Function Code 38

Entry:

a0	Address of memory block to be trimmed
d0.l	New size of the block

Exit:

xnzc			
C=0	OK	d1.l	undefined
C=1	error	d1.l	Error code

This function is used when the size of the memory block is not known. A very large memory block big enough to cater for all situations can be claimed using `_Malloc` then it can be trimmed down using `_Trim` when the actual size has been determined.

6.40. `_UnBack`

Function Code 37

Entry:

a0	Address of the memory block previously allocated and backed
----	---

Exit:

xnzc			
C=0	OK	d1.l	undefined
C=1	error	d1.l	Error code

This function returns a block of backed memory to the volatile state so that it will be lost next time the system is reset.

Note:

This function does not de-allocate the RAM.

6.41. **_UnFix**

Function code 31

Entry:

a0 A pointer to the name of a module
d0 The type of the module

Exit:

xnzvC
C=0 OK a1 points to old header (\$4AFC)
 d1.l undefined
C=1 error d1.l Error code (e.g. battery backed)

This function is used to remove a module from the system and de-allocate its memory. It first checks the module directory to make sure that the module is part of the system. If it is it checks that it has the correct module type and if so that the link count for that module is zero. If all of these conditions are met the module is deleted from the module directory.

6.42. **_UsrRam**

Function code 22

Entry:

none

Exit:

xnzvc
a0 points to the start of the RAM
d1.l size (00000400h)

There is a 1K (400h) byte section of RAM which is guaranteed not to be touched by the operating system at any time. The address of this is returned is a0 and its size in d1.

6.43. **_Vector**

Function code 40

Entry:

none

Exit:

xnzvc

a0 points to the start of the RAM vector table

When a 68000 CPU starts up it uses the reset vector found at the start of the PROMs. This is also the start of a vector table for all the interrupts and traps. After reset, this table is copied into RAM where it can be modified if required. The 68000 then uses this RAM copy for its operation. This call returns the start address of this table.

6.44. **_Wait**

Function code 9

Entry:

none

Exit:

xnzvC

C=0 OK d1.l Message

C=1 error d1.l Error code

This function moves the calling process from the running queue to the waiting queue where it will stay until it gets a signal.

6.45. **_Write**

Function Code 16

Entry:

d0.l Path number to use for output
d1.l Number of bytes to transmit
a0 Address of data to be transmitted

Exit:

xnzvC
C=0 OK d1.l The number of bytes transmitted
C=1 error d1.l Error code

This function writes a block of binary data to a device or a file. None of the control characters are recognised. If `_Write` is used on a file the file size will be expanded as required.

6.46. **_WrTick**

Function Code 20

Entry:

d0.l New counter value

Exit:

xnzvc
d1.l undefined

This function is used to set the system's elapsed time counter which it constantly being incremented at 1/100 second intervals.

7. Minos Error Codes

All of the Minos function calls return errors in the same way allowing quick and simple tests to be made by the application programs. If a Minos function returns with the carry bit clear in the CPU status register then the function was performed successfully. If the carry bit is set there has been an error and d1 contains one of the following error codes.

7.1. Code 1 `_E$Mfull` (error 65536)

This error is returned when there is not enough memory left to satisfy a `_Malloc` request. Most of the system function can return this error as they use the memory allocation functions to expand their internal data areas.

7.2. Code 2 `_E$BadSum` (error 65537)

This error is returned by `_Fix_Mod` if it finds that the module being attached is damaged in some way.

7.3. Code 3 `_E$Module` (error 65538)

This error is returned by `_Link` if the module directory does not contain a module with the correct name. It can also be returned by the I/O routines as they use `_Link` to find driver and initialisation modules.

7.4. Code 4 `_E$NoPath` (error 65539)

`_Open` and `_Create` return this error when all the I/O paths have already been opened.

Note:

This error will eventually occur if programs terminate without closing all the paths that have been opened.

7.5. Code 5 _E\$Path (error 65540)

This error is returned when an I/O function is used with an illegal path number or the path is not open.

7.6. Code 6 _E\$Mode (error 65541)

This error is returned by the I/O routines if the action they perform is not allowed by a device, e.g. the LCD driver will return _E\$Mode if _Read or _ReadLn are used on it etc.

7.7. Code 7 _E\$Esc (error 65542)

This error is returned _ReadLn if the input line was terminated by the Escape key rather than a carriage return.

7.8. Code 8 _E\$Vector (error 65543)

This error is returned by _Irq if the vector number given is not one of the interrupt vectors.

7.9. Code 9 _E\$No_Irq (error 65544)

This error is returned by the _Irq function if the polling table for the selected interrupt vector is full.

7.10. Code 10 _E\$Func (error 65545)

Drivers return this error if they do not support the requested special function. Some of the I/O calls like _Delete _SetPos and _GetPos will return this error if they are used on a driver that does not support files.

7.11. Code 11 _E\$Ready (error 65546)

This error is returned by the I/O routines if the hardware has reported that it is not ready for some reason (the door is open on a floppy drive etc.).

7.12. Code 12 _E\$Found (error 65547)

This error is returned by `_Open` if it is given a file name that does not exist.

Note:

If the driver does not exist the error will be `_E$Module` not `_E$Found`.

7.13. Code 13 _E\$Open (error 65548)

The `_Create` and `_Delete` functions return this error if the file name given currently has a path open to it.

7.14. Code 14 _E\$NetNum (error 65549)

This error will be used by the network system in the future.

7.15. Code 15 _E\$Syntax (error 65550)

This error will be used by the network system in the future.

7.16. Code 16 _E\$Known (error 65551)

This error is returned by `_Fix_Mod` if there is already a module with this name in the module directory.

7.17. Code 17 _E\$Proc (error 65552)

This error is returned whenever a signal is sent to a non-existent process. This will be used by the multi-tasking system in the future.

7.18. Code 18 _E\$Link (error 65553)

This error is returned by `_UnFix` if the module to be removed is still being used by another process. This is never happen in a single task system.

7.19. Code 19 _E\$NoBack (error 65554)

This error is returned when an attempt is made to battery back a block of RAM which is not in the battery backed area.

7.20. Code 20 _E\$BadMem (error 65555)

This error is returned by `_Dalloc`, `_Trim`, `_BackUp`, & `_UnBack` if the RAM address given does not point to an area which was previously obtained using `_Malloc`.

7.21. Code 21 _E\$Size (error 65556)

This error is returned by `_Trim` if the area of RAM being trimmed is already smaller than the required size.

7.22. Code 22 _E\$Backed (error 65557)

This error is returned when an attempt is made to deallocate an area of RAM while it is still battery backed.

7.23. Code 23 _E\$Type (error 65558)

This error is returned if the module given is not of the correct type i.e. an attempt to use `_Chain` on a data module etc.

8. Example Programs

This example shows how to use the operating system to write a string to the serial port.

```

Hello lea      Port(PC),a0  Pointer to device
                               Name
      Minos    Open        Path number in d0
      bcs.s    Bad_Dev     Branch on error
      lea      Hlo_Txt(PC),a0
      move.l   Txt_Len,d1  Minos Write
Bad_Dev
      rts
Port      dc.b  ":S0",0      Device name
Hlo_Txt   dc.b  "Hello World",10,13  ASCII
                                               string
Txt_Len   equ  *-Hlo_Txt    String length

```

The example below shows a simple terminal which simply takes all the data from the serial port and puts it on the terminal screen and takes any data from the terminal's keyboard and sends it to the serial port. Notice that the program uses the operating system to transfer as much data as it can each time, which makes very efficient use of the system's serial port buffers.

```

      lea      Port(PC),a0
      Minos    Open
      bcs.s    Bad_Dev
      move.l   d0,d7        d7 hold path number
      lea      -256(a7),a7  Make some buffer
                               space
      movea.l  a7,a0
Loop  move.l   d7,d0
      Minos    _Ready      Anything ready?
      tst.w    d1
      beq.s    None_In     Branch if not
      Minos    _Read       Read incoming chars

```

Minos

```
        move.l    #$00,d0      Terminal path number
        Minos    _Write       Write the characters
None_In
        move.l    #$00,d0
        Minos    _Ready       Anything on keyboard
        tst.w     d1
        beq.s    Loop         Branch if not

        Minos    _Read        Read the keyboard
        move.l    d7,d0        I/O path number
        Minos    _Write
        bra.s    Loop

Port    dc.b    ":S0",0
```

The final example here is for a Minos driver for a serial port based on a 68681 type serial port using an oscillator running at 14.7456MHz. Excerpts from this example are used in section 5 to illustrate how to write a driver. For details on how to write a device initialisation table for use with this driver please refer to section 5.3. This example is complete and can be assembled using your chosen cross assembler.

```
        nam      dr68681
        ttl      MINOS driver for the CPU boards

Minos macr
        trap    #00
        dc.w    \0
        endm

_MBase    set    $120000      Base address IO port
_T$Driver set    2
_Wait     set    9
_Malloc   set    0
_Dalloc   set    1
_Irq      set    12
_Signal   set    10
G_Imr     set    $4a
Ivr       set    $19
```

Imr	set	\$0b
Isr	set	\$0b
Set	set	\$1d
Clear	set	\$1f
Acr	set	\$09
Cra	set	\$05
Mra	set	\$01
Csra	set	\$03
Tba	set	\$07
Sra	set	\$03
Rba	set	\$07
_E\$Mode	set	\$06
_E\$Func	set	\$0a
G_CProc	set	\$28
_PD_Pid	set	96
_C\$CR	equ	\$0d
_DVS_Rxb	equ	\$00
_DVS_Txb	equ	\$04
_DVS_Esc	equ	\$08
_DVS_Rxi	equ	\$0c
_DVS_Rxr	equ	\$0e
_DVS_Txi	equ	\$10
_DVS_Txr	equ	\$12
_DVS_Rdy	equ	\$14
_DVS_Flg	equ	\$16
_DVS_TPid	equ	\$18
_DVS_RPid	equ	\$1a
_DVS_Port	equ	\$1c
_DVS_Drvr	equ	\$20
_DVS_Type	equ	\$24
_DVS_Irq	equ	\$26
_DVS_Baud	equ	\$28
_DVS_Parity	equ	\$29
_DVS_X	equ	\$2a
_DVS_Y	equ	\$2c
_DVS_Tks	equ	\$2e
_DVS_Sec	equ	\$2f
_DVS_Head	equ	\$30

Minos

```
_DVS_Net      equ    $31
_DVS_Flag    equ    $32
Rts_Mask     equ    $33
_DVS_Fptr    equ    $34
_DVS_Fspc    equ    $36
_DVS_Fbuf    equ    $3a
Tx_Bit       equ    $3b
Rx_Bit       equ    $3c
_DVS_TxFree  equ    _DVS_X

_type:       equ    _T$Driver

__cstart
    dc.w    $4afc          Module start marker
    dc.w    0              Space for checksum
    dc.l    0              Space for module size
    dc.l    _name-__cstart Offset to module name
    dc.l    _start-__cstart Offset to code start
    dc.w    _type          Program type
    dc.w    0
    dc.l    0
    dc.l    0
    dc.l    0
    dc.w    $a55a          Module header end marker

*****
*      Some control character and bit position
* definitions for the file server flags
*

Ctl_Chr      equ    $9b
Last_Esc     equ    0
F_Error      equ    1
F_Done       equ    2
File         equ    3
Tx_Run       equ    4
File_On      equ    7
```

```

*****
*       These are the bit numbers of the various
* interrupt status bits for the two channels
*
At_Irq      equ    0
Ar_Irq      equ    1
Bt_Irq      equ    4
Br_Irq      equ    5

Def_Par     equ    $17          Default parity
register
No_Irqs     equ    0           0 = IRQ driven

_start:
    bra    init        initialisation routine
    bra    read        read routine
    bra    write       write routine
    bra    status      special functions
    bra    dinit       deinitialisation routine

*****
*       Initialisation routine - This routine is
* responsible for setting the baud rate, data size,
* parity, lines columns etc. It must also claim
* any memory it wants to use for buffers etc.
*
*a2  -    Points at the DVS table for the device
*a3  -    The address of the chip for this device
*a6  -    Points at the system global variables
*
*a0 - a4 and d0 - d2    can be used without saving
*them
*
*       Any errors should be returned in d1 with the
* carry set
*

```

```

init: bsr          Set_Baud      Set baud rate parity
      bcs          Int_Ext
      move.l       #$300,d0      512 bytes for input &
                                output buffers
      moveq.l      #$01,d1      Non backed
      Minos        _Malloc
      bcs          Int_Ext      Exit ! error in
                                memory allocation
      move.l       a0,_DVS_Rxb(a2) Save pointer to
                                receiver buffer
      lea          256(a0),a0
      move.l       a0,_DVS_Txb(a2) Save pointer to
                                tx buffer
      lea          256(a0),a0
      move.l       a0,_DVS_Fspc(a2)

      clr.b        _DVS_Flag(a2)
      clr.w        _DVS_Fptr(a2)
      clr.w        _DVS_Rxi(a2)
      clr.w        _DVS_Rxr(a2)
      clr.w        _DVS_Txi(a2)
      clr.w        _DVS_Txr(a2)
      clr.w        _DVS_Rdy(a2)
      bset         #$07,_DVS_TPid(a2)
      bset         #$07,_DVS_RPid(a2)
      move.w       #$ff,_DVS_TxFree(a2)

      move.l       a3,d0
      bclr         #$04,d0
      movea.l      d0,a1
      beq.s        Init_A      Using channel A

      move.b       #Bt_Irq,Tx_Bit(a2)
      move.b       #Br_Irq,Rx_Bit(a2)
      ori.b        #$30,G_Imr(a6)
      move.b       #$02,Rts_Mask(a2)
      bra.s        Int_Irq

```

```

Init_A:
    move.b    #At_Irq,Tx_Bit(a2)
    move.b    #Ar_Irq,Rx_Bit(a2)
    ori.b     #$03,G_Imr(a6)
    move.b    #$01,Rts_Mask(a2)

Int_Irq:
    Ifeq      No_Irqs
    lea       Ser_Srv(pc),a0    Get service
                                routine address

    move.w    _DVS_Irq(a2),d0
    Minos     _Irq
    bcs.s     Int_Ext
    move.b    d0,Ivr(a1)        Set interrupt
                                vector number
    move.b    G_Imr(a6),Imr(a1) Put the mask
                                into the 68681

    endc

    ori.w     #$88d8,_MBase+$24
    move.b    #$01,Set(a3)
    move.b    #$80,Acr(a1)
    move.b    #$05,Cra(a3)

Int_Ext:
    rts

*****
* Initialise the 68681 for baud rate parity
* start & stop bits etc.
*
* Passed: (a3) = Base address of this port
*         (a2) = DVS pointer

Set_Baud:
    move.b    #$10,Cra(a3)      Set port back
                                to Mral
    move.w    #Def_Par,d3       Set default
                                parity

```

Minos

```
    move.b    _DVS_Parity(a2),d0
    move.b    d0,-(a7)

    lsr.b     #$01,d0           Set carry if
                                parity is on
    bcc.s     Not_Par          Branch if no
                                parity
    bclr      #$01,d3           Set parity mode
                                on
    lsr.b     #$01,d0           Set carry if
                                parity is even
    bcc.s     Not_Par
    bclr      #$02,d3           Set even parity

Not_Par:
    move.b    (a7)+,d0          Recover PD_PAR
    lsr.b     #$02,d0          Get bits/char
    andi.b    #$03,d0
    eor.b     d0,d3            Mask them into
                                mar1 data
    move.b    d3,Mra(a3)        Set parity &
                                number of bits

    move.b    _DVS_Parity(a2),d0
    lsr.b     #$04,d0
    andi.w    #$03,d0
    lea      Chr_Tab(pc),a0    Get stop bits
                                data table
    move.b    (a0,d0.w),Mra(a3) set Mra2 to no
                                of stops

    move.b    _DVS_Baud(a2),d0
    andi.w    #$000f,d0
    lea      Bau_Tab(pc),a0    Get address of
                                baud rate table
    cmpi.b    #$ff,0(a0,d0.w)  Legal baud rate
    beq.s     BadRate
    move.b    (a0,d0.w),Csra(a3) Set the baud
                                rate
```

```
        rts

BadRate
        move.l    #_E$Mode,d1
        move.w    #$01,ccr
        rts

*****
* These two tables contain data for the baud
* rate and the number of stop bits
*

Chr_Tab:    dc.b    $17
            dc.b    $18
            dc.b    $1f
            dc.b    $0

Bau_Tab:    dc.b    $ff        50
            dc.b    $ff        75
            dc.b    $ff        110
            dc.b    $ff        150
            dc.b    $00        300
            dc.b    $33        600
            dc.b    $44        1200
            dc.b    $ff        1800
            dc.b    $55        2400
            dc.b    $ff        3600
            dc.b    $66        4800
            dc.b    $88        9600
            dc.b    $99        19200
            dc.b    $bb        38400

*****
*   Read an unformatted block of data from the
*   interrupt buffer on the serial port. If we run
*   out of characters then we wait for more to come
*   in. Errors on the serial port are ignored at the
*   moment
*
```

Minos

```
*      Entry
*
*      a0 => Pointer to data buffer
*      a2 => Device Storage Space
*      d1.w Number of bytes to fetch
*
*      Returns
*
*      d1.w Number of bytes read
*      d1.w Error with carry set
*
*
*      ifne      No_Irqs

read:  movem.l  d1/a0,-(a7)
       subi.w  #$01,d1
read1  btst    #$00,Sra(a3)
       beq.s   read1
       move.b  Rba(a3),(a0)+
       dbra   d1,read1
       movem.l (a7)+,d1/a0
       rts
       endc
       ifeq    No_Irqs

read:  movem.l  d0-d3/a0,-(a7)
       movea.l  _DVS_Rxb(a2),a4      Address of
                                     buffer
       move.w   sr,d2
       ori.w   #$700,sr
       move.w   sr,d3
       bra.s   FirstRxd

NextRead:
       tst.w   _DVS_Rdy(a2)        Any bytes in
                                     the buffer ??
       bne.s   DataReady          Round again if
                                     not
       move.l  G_CProc(a6),a1
       move.w  _PD_Pid(a1),_DVS_RPid(a2)
```

```

        move.w    d2,sr
        Minos    _Wait

DataReady
        move.w    d2,sr
        move.w    _DVS_Rxr(a2),d0    Receiver remove
                                      pointer
        move.b    (a4,d0.w),(a0)+    Read the data
        addi.b    #$01,_DVS_Rxr+1(a2)Inc pointer
        move.w    d3,sr                Mask interrupts
        subq.w    #$01,_DVS_Rdy(a2)  One less byte
                                      ready
        cmpi.w    #$0a,_DVS_Rdy(a2)  Low enough for
                                      RTS
        bge.s    FirstRxd            Branch if not
        move.l    a3,d0
        bclr     #$04,d0
        move.l    d0,a3
        move.b    Rts_Mask(a2),Set(a3)

FirstRxd
        dbra     d1,NextRead Got all the chars yet
        move.w    d2,sr
        movem.l  (a7)+,d0-d3/a0
        rts

        endc

*****
*       Write a block of data to the serial port.
*       This routine must ensure that the interrupts are
*       enabled and disabled as the buffers fill up and
*       empty
*           a0 - Address of the data to send
*           a2 - DVS pointer for this device
*           a6 - System variables
*           d1 - Number of bytes to send
*
*       may destroy a0 - a4 and d0 - d2

```

Minos

```
        ifne No_Irqs

write:
    movem.l    a0/d1,-(a7)
    subq.w     #$01,d1
writel
    btst       #$02,Sra(a3)
    beq.s      writel
    move.b     (a0)+,Tba(a3)
    dbra       d1,writel
    movem.l    (a7)+,d1/a0
    rts
    endc

        ifeq No_Irqs

write:
    movem.l    d0-d3/a0,-(a7)
    movea.l    _DVS_Txb(a2),a4    Address of TXD
                                         buffer
    move.w     sr,d2
    ori.w      #$700,sr
    move.w     sr,d3
    bra.s      FirstTxd

Do_Txd
    move.w     d2,sr
    tst.w      _DVS_TxFree(a2)
    bne.s      HasRoom
    move.l     G_CProc(a6),a1
    move.w     _PD_Pid(a1),_DVS_TPid(a2)
    Minos     _Wait

HasRoom
    move.w     d3,sr
    move.w     _DVS_Txi(a2),d0
    addq.b     #$01,_DVS_Txi+1(a2)
    move.b     (a0)+,0(a4,d0.w)    Put data into
```

```

                                buffer
    subq.w    #$01,_DVS_TxFree(a2)
FirstTxd
    dbra     d1,Do_Txd          Round again for
                                next char

    bset     #Tx_Run,_DVS_Flag(a2)
    bne.s    TxEnd
    move.b   #$04,Cra(a3)      Enable Tx
    move.w   _DVS_Txr(a2),d0
    move.b   0(a4,d0.w),Tba(a3)Send the data
    addq.b   #$01,_DVS_Txr+1(a2)Move index
                                pointer up
    addq.w   #$01,_DVS_TxFree(a2)

TxEnd move.w   d2,sr
    movem.l  (a7)+,d0-d3/a0
    rts

    endc

*****
*       These routines provide the status & special
*       functions.
*
*       Passed
*
*       d1 - Function code
*       other registers depend on the function
*

status
    tst.w    d1
    bne.s    free

    ifeq    No_Irqs
    clr.l    d1
    move.w   _DVS_Rdy(a2),d1
    endc

```

Minos

```
        ifne        No_Irqs
        clr.l       d1
        btst        #$00,Sra(a3)
        beq.s       Rdy_End
        moveq.l     #$01,d1
        endc
Rdy_End
        rts

free   cmp.w       #$01,d1
        bne.s       Tri_Ink

        clr.l       d1
        move.w      _DVS_TxFree(a2),d1
        rts

*****
*       Function 2 provides non IRQ driven input
* character it returns with d2 = -1 if no char are
* available
*
Tri_Ink
        cmpi.w      #$02,d1
        bne.s       Tri_Raw

        moveq.l     #-1,d2
        btst        #$00,Sra(a3)
        beq.s       No_Key
        move.b      Rba(a3),d2
        andi.l      #$ff,d2
No_Key
        rts

*****
*       Function 3 provides a non IRQ driven outchar
* which is used by the debugger
*
```

```
Tri_Raw
    cmpi.w    #$03,d1
    bne.s     Tri_Opn

Wr_Lop
    move.w    sr,-(a7)
    ori       #$700,sr
    bset      #Tx_Run,_DVS_Flag(a2)    Set if TX
                                           is already running
    bne.s     Tx_On
    move.b    #$04,Cra(a3)             Enable TX
    move.b    d2,Tba(a3)
    nop
    move.w    (a7)+,sr
    rts

Tx_On btst    #$02,Sra(a3)
    beq.s     Tx_On
    move.b    d2,Tba(a3)
    move.w    (a7)+,sr
    rts

*****
*   Take care of open, close and create status
*   calls

Tri_Opn
    cmpi.w    #$10,d1
    beq.s     nul_stat
    cmpi.w    #$11,d1
    beq.s     nul_stat
    cmpi.w    #$12,d1
    bne.s     S_File
nul_stat
    clr.l     d1
    rts

    include sfile.asm
```

Minos

Not_Sfile

```
    move.l    #_E$Func,d1
    move.w    #$01,ccr
    rts
```

dinit

```
    cmpi.w    #$ff,_DVS_TxFree(a2)
    bne.s     dinit
    movea.l   _DVS_Port(a2),a3    Chip address
    move.b    Tx_Bit(a2),d0
    bclr      d0,G_Imr(a6)
    move.b    Rx_Bit(a2),d0
    bclr      d0,G_Imr(a6)
    move.b    G_Imr(a6),Imr(a3)  Disable this
                                chanel IRQs

    movea.l   #$00,a0
    move.w    _DVS_Irq(a2),d0
    Minos    _Irq                Remove our IRQ
                                routine

    movea.l   _DVS_Rxb(a2),a0
    Minos    _Dalloc
    rts
```

```
*****
*      Interrupt service routine. This service
* routine must poll
* the device to make sure that the IRQ is for us
* then buffer it in either the file server input of
* the character input buffer
*
*          a2 - Pointer to DVS
*
*          may destroy a1 a2 a3 d0 & d1
*
```

Ser_Srv

```
        move.l    _DVS_Port(a2),d1  Chip address
        move.l    d1,a3
        bclr     #$04,d1
        movea.l   d1,a1
        move.b    Rx_Bit(a2),d0
        btst     d0,Isr(a1)
        bne.s    My_Rxd
        move.b    Tx_Bit(a2),d0
        btst     d0,Isr(a1)
        bne.s    My_Txd
        move.w    #$01,CCR
Irq_Ext
        rts

*****
*       Start of transmitter service
*
My_Txd
        cmpi.w    #$ff,_DVS_TxFree(a2)
        bne.s    Can_Go

*****
*       Reaches here if the transmitter has
* interrupted but there are no characters to go. We
* disable the transmitter to clear it's IRQ status
*
        bclr     #Tx_Run,_DVS_Flag(a2)  Mark
                                           transmitter idle
        move.b    #$08,Cra(a3)
        bra.s    Tx_Iext                Exit

Can_Go
        movea.l   _DVS_Txb(a2),a1      Transmitter
                                           buffer address
        move.w    _DVS_Txr(a2),d0
        move.b    0(a1,d0.w),Tba(a3)  Send the data
        addq.b    #$01,_DVS_Txr+1(a2) Move index
                                           pointer up
```

Minos

```
    addq.w    #$01,_DVS_TxFree(a2)
    cmpi.w    #$c0,_DVS_TxFree(a2)
    blt.s     Tx_Iext
    move.w    _DVS_TPid(a2),d0
    bmi      Tx_Iext
    clr.l     d1
    Minos    _Signal
    Bset     #$07,_DVS_TPid(a2)
Tx_Iext
    rts

*****
*      Incomming serial interrupt arrives
* here

My_Rxd
    move.b    Rba(a3),d0
    btst     #File_On,_DVS_Flag(a2)  Is the
                                     file server enabled
    beq.s    SIrq                    Straight to serial
                                     input if not
    btst     #File,_DVS_Flag(a2)    Set if
                                     were loading a file
    bne     File_Irq                Service as if we are
                                     a disk drive

*****
*      Reaches here if a serial interrupt is
* received and we are in character input mode.

Ser_Irq
    cmpi.b    #Ctl_Chr,d0  Start of a control
                           sequence ??
    bne.s    Not_SCtl     Branch to serial
                           service if not

    btst     #Last_Esc,_DVS_Flag(a2) Set if
                                     last char was CTL as well
    bne.s    SIrq        Go and put one CTL char in
```

```

                buffer
    bset         #Last_Esc,_DVS_Flag(a2) Say this
                one is a CTL char
    bra         Irq_Ext      Exit

Not_SCTl
    Bclr        #Last_Esc,_DVS_Flag(a2) Say this
                char is not CTL
    beq.s       SIrq        Normal input if last
                was 'nt either
    bset        #File,_DVS_Flag(a2)      Say we
                have started loading
    tst.b       d0          $00 = File ok
                $80 = File bad
    beq         Irq_Ext      Exit
    bset        #F_Error,_DVS_Flag(a2) Mark
                error in file
    bra         Irq_Ext      Exit

Sirq  move.w    _DVS_Rxi(a2),d1
      addq.b    #$01,d1
      cmp.w    _DVS_Rxr(a2),d1    Is there room
                                for the char
    beq         Irq_Ext          No room exit
                                without char

      cmpi.w    #$f0,_DVS_Rdy(a2) High enough for
                                RTS yet ??
    blt.s      No_Rts1

      move.b    Rts_Mask(a2),Clear(a1)

No_Rts1
    movea.l    _DVS_Rxb(a2),a1    Point at buffer
                                base
    move.w    d1,_DVS_Rxi(a2)    Update pointer
                                for next time
    subq.b    #$01,d1
    addq.w    #$01,_DVS_Rdy(a2)

```

Minos

```

    move.b    d0,0(a1,d1.w)
    tst.l     _DVS_Esc(a2)      Is there anyone
                                for ESC
    beq.s     NoEscape         Exit if not
    cmpi.b    #27,d0           Was it an
                                escape character ??
    bne.s     NoEscape         Exit if not
    movea.l   _DVS_Esc(a2),a3
    move.w    #-1,(a3)         Signal Escape

NoEscape
    move.w    _DVS_RPid(a2),d0
    bmi      Irq_Ext
    clr.l     d1
    Minos    _Signal
    bset     #$07,_DVS_RPid(a2)
    bra      Irq_Ext

*****
*   Reaches here if we are in the middle of a
*   file download

File_Irq
    cmpi.b    #Ctl_Chr,d0      Is it a control
                                character
    bne.s     Not_Fctl        Branch if not

    bclr     #Last_Esc,_DVS_Flag(a2) Was the
                                last char a CTL
    bne.s     FIrq           Go and insert one CTL
                                character
    bset     #Last_Esc,_DVS_Flag(a2) Mark this
                                CTL character
    Beq      Irq_Ext         always

Not_Fctl:
    cmpi.b    #$0d,d0         End of a file
                                operation ??
    bne.s     Not_FEnd
```

```

        bset      #F_Done,_DVS_Flag(a2)
        bclr     #File,_DVS_Flag(a2)
        bra      Irq_Ext

Not_Fend
        bclr     #Last_Esc,_DVS_Flag(a2) Was the
                                last one a CTL char
        beq.s    FIrq      Normal insert if not
        move.b   #$0d,d0
FIrq    tst.l    _DVS_Fbuf(a2)
        Beq      Irq_Ext
        movea.l  _DVS_Fbuf(a2),a1  Get file buffer
                                address
        move.w   _DVS_Fptr(a2),d1
        move.b   d0,(a1,d1.w)
        addi.w   #$01,_DVS_Fptr(a2)
        bra      Irq_Ext

_name dc.l 0      Space for module name text

```

The file sfile.asm is listed below.

```

_SF_Read    set    $80
_SF_ReadLn  set    $81
_SF_Write   set    $82
_SF_Open    set    $83
_SF_Create  set    $84
_SF_Close   set    $85
_SF_Seek    set    $86
_SF_Tell    set    $87
_SF_Delete  set    $88
_SF_Size    set    $89
_SF_Enable  set    $8a
_SF_Disable set    $8b

```

```

*****
*      Fread call to support serial fileing system.
* The first two services are used to turn the file
* server functions on or off this allows serial

```

Minos

* ports to be used for binary input if the file
* functions are not required.
*

S_File:

```
    cmpi.w    #_SF_Enable,d1
    bne.s     Not_Ena

    bset      #File_On,_DVS_Flag(a2)
    move.l    #$00,d1
    rts
```

Not_Ena

```
    cmpi.w    #_SF_Disable,d1
    bne.s     Not_Dis
    bclr      #File_On,_DVS_Flag(a2)
    move.l    #$00,d1
    rts
```

Not_Dis

```
    btst      #File_On,_DVS_Flag(a2)   File
                                         server enabled ?
    bne.s     File_Svc   Branch if it is
    move.l    #_E$Mode,d1
    move.w    #$01,ccr
    rts
```

* Main file server routines. It can only reac
* here if the file server is enabled on this device
*

File_Svc

```
    cmpi.w    #_SF_Read,d1
    bne.s     Not_Read
    move.b    #24,1(a0)   Read block command
    bra.s     Do_Read
```

Not_Read

```

        cmpi.w    #_SF_ReadLn,d1
        bne.s    Not_Rdln
        move.b   #25,1(a0)
Do_Read
        move.l   a0,_DVS_Fbuf(a2)  Save pointer to
                                   file buf
        clr.w   _DVS_Fptr(a2)
        bclr   #F_Error,_DVS_Flag(a2)
        bclr   #F_Done,_DVS_Flag(a2)

        move.b   #27,0(a0)
        move.b   d2,2(a0)          Disk path no
        move.b   d3,3(a0)          Block size
        move.w   #$04,d1
        move.l   a0,-(a7)
        bsr     write              Send Read line
                                   message
        move.l   (a7)+,a0
        bsr     _Fwait
        bcs.s   Rd_Ext
        move.w   _DVS_Fptr(a2),d1
Rd_Ext
        rts

*****
*      Reach here after a serial filing system
* command has been sent. We wait here until we get
* a response from the host then we pick up any
* errors and return

_Fwait
        bclr   #F_Done,_DVS_Flag(a2)Operation
                                   finished yet ??
        beq.s   _Fwait            Round again if not
        clr.l   _DVS_Fbuf(a2)
        clr.l   d1
        move.b   (a0),d1
        bclr   #F_Error,_DVS_Flag(a2)  Any error

```

Minos

```
                beq.s      No_Ferr      in this operation
                move.w     #$01,ccr     branch if not
No_Ferr
                rts
```

```
*****
```

```
*
```

```
*      Create a file on the host PC
```

```
Not_Rdln
                cmpi.w     #_SF_Create,d1
                bne.s      F_Open

                move.b     #30,d0
                bsr        Set_Cmd
                bra.s      Nam_Lop
```

```
*****
```

```
*      Fopen call tries to open a file on the host
```

```
F_Open
                cmpi.w     #_SF_Open,d1
                bne.s      F_Close

                move.b     #22,d0
                bsr        Set_Cmd      Set command block
                                        without file ID
```

```
Nam_Lop
                tst.b      (a0)
                beq.s      Nam_Dun
                move.b     (a0)+,(a1)+
                addq.w     #$01,d1
                bra.s      Nam_Lop
Nam_Dun
```

```
    move.b    #_C$CR,(a1)
    addq.w   #$01,d1
    movea.l  _DVS_Fbuf(a2),a0
    bsr     write
    movea.l  _DVS_Fbuf(a2),a0
    bsr     _Fwait
    rts
```

```
*****
*      Fclose called each time a path is closed. We
* must pass on the close to the host so that it
* will close files a that end
```

F_Close

```
    cmpi.w   #_SF_Close,d1
    bne.s    Not_Clo

    move.b   #23,d0      Close file command
    bsr     Set_Cmd1    Set up command block
                          with file ID
    movea.l  _DVS_Fspc(a2),a0
    bsr     write
    movea.l  _DVS_Fspc(a2),a0
    bra     _Fwait
```

```
*****
*      d2.w   file ID number
*      d3.w   Size of data block
```

Not_Clo

```
    cmpi.w   #_SF_Write,d1
    bne     F_Seek
```

F_Write

```
    move.l   d3,-(a7)
    move.b   #26,d0
    bsr     Set_Cmd1
    subi.w   #$01,d3      Don't loop too much
    bmi.s   Fw_Exit      Tried to write an
```

empty block

```
*****
*      Block header is set we now start putting data
* into the buffer replacing $9b with $9b9b and $0d
* with $9b2d
```

```
Fw_Loop
    move.b      (a0)+,d0      Get next byte from
                             input
    cmpi.b     #_C$CR,d0     Is it a <CR> ??
    bne.s      Not_Cr
    move.b     #$9b,(a1)+    Special control start
    addq.w     #$01,d1       Say one more char in
    move.b     #$2d,d0       Replace the $0d with
                             $2d
    bra.s      Fw_Out        Insert it as normal

Not_Cr
    cmpi.b     #$9b,d0       Transmitting the
                             control char ??
    bne.s      Fw_Out        No then insert as
                             normal
    move.b     d0,(a1)+      Put an extra control
                             char in
    addq.w     #$01,d1       Say one more in

Fw_Out
    addq.w     #$01,d1
    move.b     d0,(a1)+
    dbra      d3,Fw_Loop

Send_It
    move.b     #_C$CR,(a1)+   Put on a real
                             CR to show end
    addq.w     #$01,d1
    movea.l    _DVS_Fspc(a2),a0 a0 Points at
                             start of block
    bsr       write          Send it to host
```

```
                                system
    movea.l    _DVS_Fspc(a2),a0
    bsr       _Fwait
Fw_Exit
    movem.l   (a7)+,d3
    rts

*****
*           Seek a file to a new position
*
*           d2.w - File Number
*           d3.l - File Position

F_Seek
    cmpi.w    #_SF_Seek,d1
    bne      F_Tell

    move.b    #28,d0
    bsr      Set_Cmd1

    move.w    #$03,d2
Seek_Lop
    move.b    d3,(a1)+
    asr.l    #$08,d3
    addq.w   #$01,d1
    dbra     d2,Seek_Lop
    movea.l   _DVS_Fspc(a2),a0
    bsr      write
    movea.l   _DVS_Fspc(a2),a0
    bsr      _Fwait

    move.l    2(a0),d1
    rts

F_Tell
    cmpi.w    #_SF_Tell,d1
    bne      Not_Tell

    move.w    #29,d0
```

Minos

```
        bsr          Set_Cmd1
        movea.l      _DVS_Fspc(a2),a0
        bsr          write
        movea.l      _DVS_Fspc(a2),a0
        bsr          _Fwait
        bcs.s        Tel_Ext
        movea.l      _DVS_Fspc(a2),a0
        move.l       (a0),d1
Tel_Ext
        rts

*****
*      Delete a file

Not_Tell
        cmpi.w       #_SF_Delete,d1
        bne.s        File_Size

        move.w       #31,d0          Delete file in PC
                                       command

        bsr          Set_Cmd
        bra          Nam_Lop        Use filename copy
                                       from 'open'

*****
*      Determine the size of a file

File_Size
        cmpi.w       #_SF_Size,d1
        bne          Not_Sfile
        move.w       #33,d0
        bsr          Set_Cmd1
        movea.l      _DVS_Fspc(a2),a0
        bsr          write
        movea.l      _DVS_Fspc(a2),a0
        bsr          _Fwait
        bcs.s        Size_Ext
        movea.l      _DVS_Fspc(a2),a0
        move.l       (a0),d1
```

```
Size_Ext
    rts
```

```
*****
```

```
Set_Cmd1
    bsr.s      Set_Cmd
    move.b    d2,(a1)+
    addq.w    #$01,d1
    rts
```

```
Set_Cmd
    clr.w     _DVS_Fptr(a2)
    bclr     #F_Error,_DVS_Flag(a2)
    bclr     #F_Done,_DVS_Flag(a2)    Set up
                                     flags to get error block
    movea.l   _DVS_Fspc(a2),a1    Use file buffer
                                     to build block
    move.l    a1,_DVS_Fbuf(a2)    Also used to
                                     get returning status
    move.b    #27,(a1)+
    move.b    d0,(a1)+
    move.w    #$02,d1
    rts
```