1. INTRODUCTION

EXOS is the extendable operating system for the ENTERPRISE micro-computer. It provides an interface between an applications program (such as the IS-BASIC interpreter) and the hardware of the machine. The main features of EXOS are a channel based input/output system and sophisticated memory management facilities. The I/O system allow device independent communication with a range of built in devices and also any additional device drivers provided by the user.

The built in devices included with the EXOS kernel in the ENTERPRISE ROM, are:

1. Video driver providing text and graphics handling.
2. Keyboard handler providing joystick, autorepeat and programmable function keys.
3. Screen editor with word processing capabilities.
4. Comprehensive four source stereo sound generator.
5. Cassette tape file handler.
6. Centronics compatible parallel interface.
7. RS232 type serial interface.
8. Intelligent Net three wire network interface.

This document describes the EXOS kernel, which interfaces between an applications program and the various devices, providing memory management and various other facilities. It explains the action of the kernel from the point of view of both devices and applications programs. The built in device drivers themselves are each described in separate documents, some of which make reference to the kernel specification.

It is intended that, along with the various device driver specifications, this document will provide sufficient information for writing applications programs using EXOS, or for writing new EXOS device drivers. All details in this document apply to EXOS version 2.0.

ADQUIPMENT BV
INDUSTRIEWEG 10-12
POSTBUS 311
3440 AH WOERDEN
TEL. 03480-18341

5

2. OVERVIEW OF THE EXOS ENVIRONMENT

When EXOS is running, there is always a "current applications program" which has overall control of the machine. This program can call EXOS to make use of any of its facilities, such as channel I/O or memory allocation. In the standard machine the current applications program will be either the built in word processor (WP) program or the IS-BASIC interpreter cartridge, although it could be any other cartridge ROM or cassette loaded program in RAM.

Throughout this document the term "user" is used to refer to the current applications program, since this program is using EXOS.

2.1 The EXOS Input/Output system

As mentioned before, the EXOS I/O system is provided as a set of device drivers. A device driver is a piece of code containing all the necessary routines to control the device it is serving, and provide a standard interface to EXOS. A device driver might not in fact control a physical device but may provide device-like facilities such as reading and writing characters, purely in software.

When EXOS starts up it locates all the built in device drivers and makes an internal list of them. The list also includes device drivers contained in any expansion ROMs which are plugged in. The user can link in additional devices (known as user devices) which are added to the list. Each device in the list is identified by a device name such as "VIDEO", "NET" or "KEYBOARD".

The I/O system is channel based, which means that in order to communicate with a device, a channel must first be opened. A channel is opened by giving the device name and a one byte channel number to EXOS. This establishes a communications path to the device along which characters can be transferred in either direction, either singly or in arbitrarily-sized blocks, and special commands given to the device, simply by specifying the channel number.

For a file based device (such as cassette tape or disk) a channel would be opened to do a single file transfer and then closed again. For non-file devices (such as the keyboard) a channel would probably be opened and then remain open for all future accesses.

EXOS allows many channels to be opened simultaneously to a single device, although some devices themselves will not allow this. For example the video driver allows any number of channels open to it but the keyboard driver allows only one. Channels remain open until they are explicitly closed by the user.

When a channel is opened, EXOS takes care of allocating any RAM which the device might need for buffers or variables.

2.2 Memory Allocation

In order to understand the memory allocation facilities of EXOS it is first necessary to understand the hardware memory organisation on the Enterprise.

2.2.1 Memory Segments and Pages

The Enterprise uses a segmented memory scheme in order to extend the addressing capability of the Z-80 from 64 kilobytes to 4 megabytes. The segmenting scheme is based on 16k segments.

The Z-80 address space is divided up into four 16k "pages", numbered from zero to three. The addresses for these four pages are:

Page-0	0000h - 3FFFh
Page-1	4000h - 7FFFh
Page-2	8000h - BFFFh
Page-3	C000h - FFFFh

The 4 megabyte address space is divided up into 256 "segments", each segment being 16k. Every 16k section of memory in the system thus has its own "segment number" in the range [00h - FFh]. The segment numbers for certain sections of memory are permanently defined:

Internal 32k ROM	-	Segments 00h and 01h
64k Cartridge slot	-	Segments 04h to 07h
Internal 64k RAM	-	Segments FCh to FFh
2nd internal 64k RAM	-	Segments F8h to FBh

Associated with each of the four Z-80 pages there is an 8-bit "page register" on a Z-80 I/O port. The contents of these registers define which of the 256 possible segments are to be addressed in each of the Z-80 pages. Thus any segment can be addressed in any of the Z-80 pages simply by putting its segment number into the appropriate page register. One segment can be simultaneously addressed in two or more pages if desired by putting the same value into several of the paging registers.

The four internal RAM segments (segment numbers FCh to FFh) are the only ones which the NICK chip can address for generating video displays. For this reason they are referred to as the video RAM. They are also slower to access than all other memory since any Z-80 accesses to them are subject to clock stretching to sychronise with the NICK chip accesses.

2.2.2 User Segment Allocation

When EXOS starts up it locates and tests any RAM segments which are available and builds up a list of them. When it passes control to the user, it will do so by putting the appropriate segment (usually a ROM segment) into Z-80 page-3 and jumping to it. At this stage the contents of pages 1 and 2 will be undefined, but page-0 will contain a RAM segment, known as the "page zero segment".

The first 256 bytes of the page zero segment contain certain system entry points and system code, and also certain areas which are reserved for CP/M emulation. The rest of the page zero segment is not used by the system and is completely free for use by the user. Because of the system entry points, which include an interrupt entry point, the page zero segment should always be kept in Z-80 page-0.

If the user requires more RAM then it can ask for additional segments from EXOS. It will be allocated other RAM segments from the list unless there are none left. It can also free a segment which it has been allocated when it does not need it any more. These additional segments will not be explicitely paged in by EXOS, it is up to the user to page them in (usually into pages 1 and 2) when it needs them.

It is possible for the user to be allocated a "shared segment". This is a segment of which the user is only allowed to use part, the rest being used by EXOS. This will be explained in more detail later.

2.2.3 EXOS RAM usage and Channel RAM

Segment number 0FFh; which is one of the video RAM segments, is always used by EXOS and is therefore known as the "System segment". The details of what this segment is used for will be given later but it includes RAM areas for system variables, system stack, built in device driver variables, line parameter table, lists of RAM and ROM segments, the list of available devices and RAM allocation for extension ROMs. These RAM areas start at the top of the segment and use as far down as necessary.

Below this system RAM allocation is the channel RAM area. This contains an area of RAM for every channel which is currently open. The size of each RAM area is determined by the device when the channel is opened and may be any size from just a few bytes up to several kilobytes. These channel RAM areas always start in the system segment but can occupy any number of other segments. The RAM for any given channel is de-allocated when the channel is closed so this memory allocation is not permanent.

2.3 System Extensions (ROM and RAM)

When EXOS starts up, as well as making a list of all available RAM, it also looks for any extension ROMs which are plugged in and builds up a list of these. Each of these ROMs may contain EXOS device drivers which will be linked into the system just like built in devices. Each ROM also contains an entry point which is used for several purposes.

Each ROM will be given a chance to become the current applications ROM at startup time. If no ROM takes up this opportunity then the internal word processor will take control.

At certain times an "extension scan" will be done which gives each ROM in the list a chance to carry out some service. This allows ROMs to provide additional error messages, help messages and various other system functions. An extension scan can be initiated by the user program which will pass a command string to each ROM in turn. This allows an extension ROM to provide some service or carry out a command and then return to the main applications ROM. This facility can also be used to start up another ROM as the current applications program.

There is a facility in EXOS for the system to load programs into system RAM (ie. RAM which is not allocated to the user) and link these into the list of ROMs. Thus all the facilities which are available to extension ROMs are also available to code loaded into RAM. These RAM extensions can be loaded either into a complete 16k segment each, or if they are supplied in a relocatable format, several of them can be put into one segment thus reducing the amount of RAM which is used up in this way.

3. SYSTEM INITIALISATION and WARM RESET

3.1 Cold Reset Sequence

A cold reset is done when the machine is first powered on, and when the RESET button is pressed, unless the user has set up a "warm reset address" (see below). It completely restarts the system, losing any information which existed before the reset.

A cold reset first does a checksum test of the internal 32k ROM. If this is passed it then locates any RAM in the system. It searches the whole 4-megabyte address space apart from the internal ROM and cartridge slot (segments 00 to 07). It examines each 16k segment in turn, doing a memory test on each one. If a segment passes the memory test then it will be added to the list of available RAM segments. There is no test for RAM reflections so any extension RAM must be decoded fully. The memory test destroys any data which may have been in the RAM segment previously.

After the RAM test, the 4-megabyte memory space is then searched for extension ROMs. The ROM search will only find ROMs in segment numbers which are multiples of 16. This means that extension ROMs have to be decoded only to 256k boundaries, but can reflect throughout this 256k space. An exception is made for the cartridge slot in that all four segments are examined for ROM, but a test is done to ignore reflections by checking that any two ROMs in the cartridge slot are different. The details of extension ROMs are explained later.

Having created the ROM list, various internal variables are set up, including the system entry points at the start of the page zero segment. The remainder of the I/O system is then initialised by linking in and initialising all the built in and extension devices and initialising all extension ROMs as will be explained in more detail later on. The copyright display program is then entered which displays a flashing "ENTERPRISE" message and an Intelligent Software copyright message on the screen, until a key is pressed by the user. This display and waiting for a key can be suppressed by an extension ROM setting the variable CRDISP_FLAG to a non-zero value when it is initialised (see below for ROM initialisation).

When a key is pressed, the display will be removed and the system will call each extension ROM in turn with action code 1 (see later for explanation of action codes). Any ROM which wants to set itself up as the current applications program simply does an "EXOS reset" call (see later) to claim the system and then has full control.

3.2 Warm Reset Sequence

A warm reset is performed when the RESET button on the machine is pressed, if the user has set up a warm reset address, and if the system variable area has not been corrupted. A warm reset address can be set up simply by storing the address in the variable RST_ADDR which is in a defined place in the system segment. The address stored must be in Z-80 page-0 and will be jumped to when the warm reset sequence is complete. The warm reset routine will thus always be in RAM since the page zero segment is RAM.

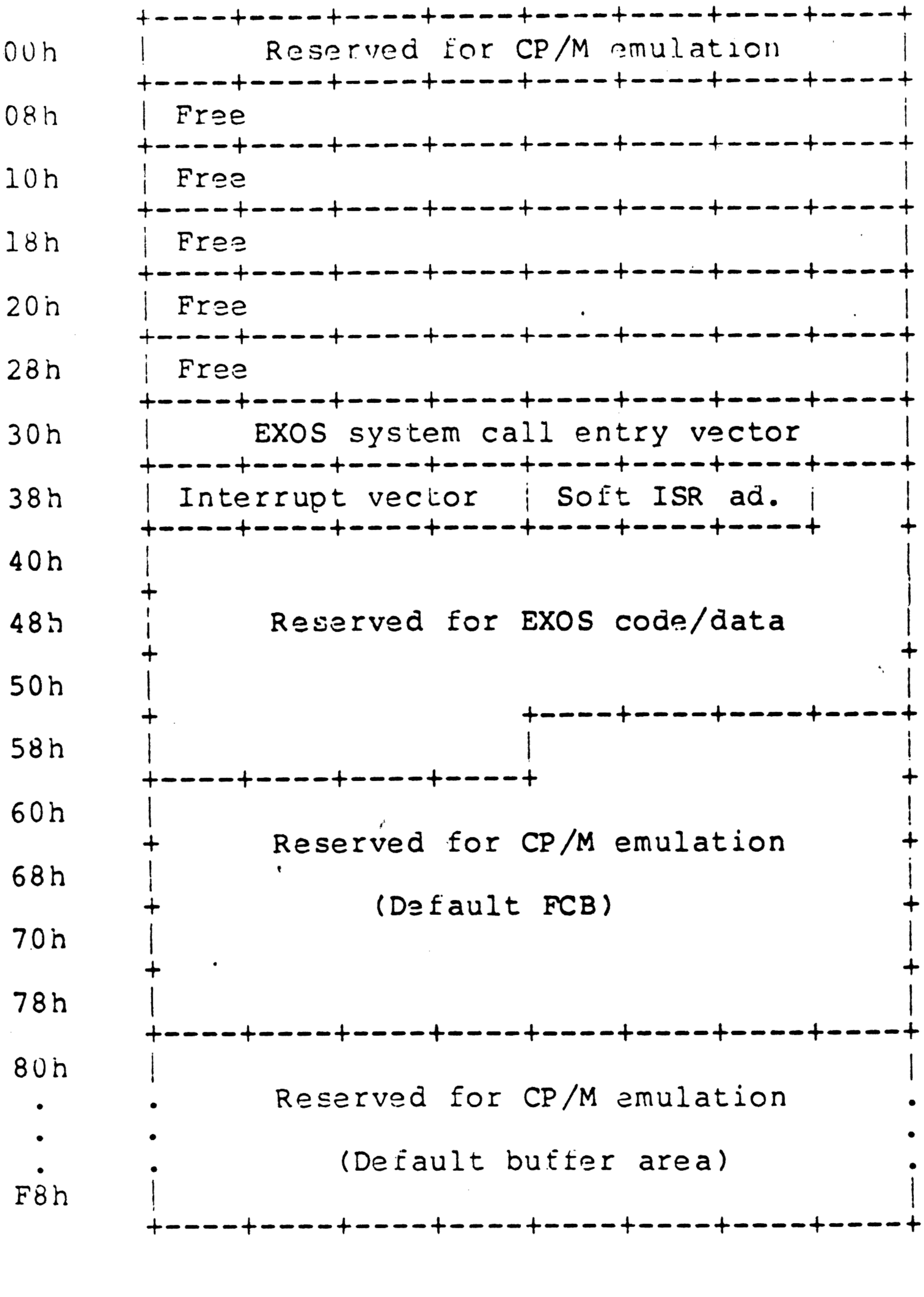
A warm reset does not do a RAM test or a ROM search. All memory allocated to the user is undisturbed and any system RAM extensions or user devices which are linked in, remain. However all channels are forcibly closed and all devices are re-initialised, any RAM which was allocated to channel RAM areas is freed. The details of this will be explained later on (in fact an "EXOS reset" call is simulated with the reset flags set to 10h - see later).

EXOS will set RST_ADDR back to zero before jumping to the warm reset address. This ensures that if the system has crashed then a second press of the reset button will do a cold reset. Also, as long as the user waits for a short time before setting its warm reset address up again, pressing the reset button twice quickly will always do a cold reset.

The code at the warm reset entry point will be entered exactly as if it had just done an "EXOS reset" call so it will have to set up its stack pointer and re-enable interrupts (see section on the "EXOS reset" call). The contents of Z-80 pages 1, 2 and 3 will be un-defined so the user must reset these for himself. Particularly, in the case of a ROM applications program which normally runs with its ROM in page-3, it will have to page its own ROM back in. This means of course that the applications program must have stored its segment number in the page zero segment in order for the warm reset routine to restore it. Also note that any software interrupt address (described later) which may have been set up will have been lost, and so this must be set up again.

4. APPLICATIONS PROGRAM INTERFACE

The first 256 bytes of the page zero segment, which always resides in Z-80 page-0, are laid out as follows.



The areas which are listed as reserved for CP/M emulation can be used by any programs which do not require CP/M compatibility, but are never used by EXOS. The system entry points will be described below.

An applications program is started up by being entered at its entry point address with a certain action code and possibly a command string (see section on scanning extensions). To take control of the system, the user must do an "EXOS reset" call with the reset flags set correctly depending on the action code (see the section on scanning extensions and also the description of the "EXOS reset" call). Having done this call, the user must set up his own stack and then enable interrupts. It then has full control of the system.

The segment with the applications program code in, for example the cartridge ROM, will always be entered in Z-80 page-3 by EXOS and generally it is convenient to leave it permenantly in page-3, although it can be moved if desired. When an EXOS call is made, or an interrupt occurs, then contents of pages 1, 2 and 3 will be changed, possibly many times, but will always be restored to their original segments before returning to the user. Thus whatever paging the user sets up will be preserved by all EXOS calls and interrupts.

4.1 EXOS System Calls - General

An EXOS call is made by executing a "RST 30h" instruction. The area from 30h to 5Bh contains code to handle the transfer of control to the main EXOS ROM and also to handle the return to the user. This entire area should not be modified by the applications program at all, except for the software interrupt address at 3Dh and 3Eh (described later).

The different EXOS calls are defined by a one byte function code which immediately follows the "RST 30h" instruction. Parameters to the EXOS calls are passed in registers A, BC and DE, and these registers are also used to return results. Register A always returns a status value which is zero if the call was successful and non-zero if an error or unusual condition occurred. There is a function call which will provide a simple text string explanation for these status codes.

Registers AF, BC and DE will not be preserved by any EXOS calls except in certain specific cases which are noted in the detailed descriptions of the calls. The contents of all other registers, (HL, IX, IY and the alternate register set including AF'), and of the four Z-80 page registers, will be preserved by all EXOS calls, except in a few specific cases which are also noted in the detailed functional descriptions.

EXOS always switches to an internal system stack in the system segment whenever it is entered, and therefore uses very little space on the user's stack. However, at least 8 bytes should always be available beyond the top of the stack. Even if no EXOS calls are made, this space is required for interrupt servicing. The program stack should also be managed correctly such that there is never any wanted information above the stack pointer, it can be anywhere in Z-80 memory, provided it is in RAM of course.

The system calls will be explained in more detail later but here is a list of them all with their function codes.

Code	Function
0	System reset
1	Open channel
2	Create channel
3	Close channel
4	Destroy channel
5	Read character
6	Read block
7	Write character
8	Write block
9	Channel read status
10	Set and read channel information
11	Perform special function on channel
16	Read/Write/Toggle EXOS Variable
17	Capture channel
18	Re-direct channel
19	Set default device name
20	Return system status
21	Link device
22	Read EXOS boundary
23	Set user boundary
24	Allocate segment
25	Free segment
26	Scan system extensions
27	Allocate channel buffer (device only function)
28	Explain error code
29	Load module
30	Load relocatable module
31	Set time
32	Read time
33	Set date
34	Read date

Function calls 1 to 11 are device calls. They each take a channel number in register A and the call will be passed on by EXOS to the appropriate device driver for that channel. Almost all of the other functions are handled entirely within the EXOS kernel. The exceptions are: "Scan system extensions" (code 26) which is an explicit request to pass a command string around all ROM and RAM extensions, and "load module" (29), "explain error code" (28) and "read/write/toggle EXOS variable" (16) which will offer their parameters to any extensions if they are not recognised.

When a device or system extension has control as a result of one of these calls being made, it is able to make its own EXOS calls. In this way EXOS is re-entrant, although there are some limitations on this. Device drivers are not allowed to open or close channels when they have control (because of channel buffer moving problems - see later). The "allocate channel buffer" call (code 27) can only be made by a device during an open channel call, the user should never make this call.

The EXOS calls which can result in nested EXOS calls being made carry out stack checking to ensure that the internal system stack does not overflow. This effectively limits the depth of nesting allowed although there is an absolute limit of 127 levels beyond which the system will not work. It is difficult to imagine this depth of nesting being required.

4.2 Hardware and Software Interrupts

EXOS uses hardware interrupts to keep its clock/calendar up to date. Each device driver can also have an interrupt routine which EXOS will call whenever a specified type of interrupt occurs. Details of this are given with the explanation of device descriptors. There is no facility for the user to have an interrupt routine. However the user is provided with a facility for handling software interrupts.

Software interrupts provide a way for the user to be alerted to various events occuring within EXOS. A software interrupt is triggered by a device driver's interrupt routine detecting some special occurence, such as the network driver having received a block of data from the network. When this occurs the device stores a "software interrupt code" in the variable FLAG_SOFT_IRQ which is in the system segment. This code indicates what the reason for the software interrupt was.

Nothing else occurs until EXOS is about to return to the user, which may be directly from the interrupt routine or may be very much later if the interrupt occured while a device driver was executing. At this time a software interrupt will be carried out if the user has defined a non-zero "software interrupt address". This address is defined simply by storing the address at 3Dh and 3Eh in page-0, which is in fact the operand of a jump instruction.

The software interrupt is carried out by EXOS jumping to the software interrupt address (which can be in any Z-80 page) instead of executing the normal "RET" instruction which would return to the user. The environment will be exactly as it would be if the return had been made, with the correct paging and stack pointer. The return address will still be on the stack so the software interrupt routine may return to the main program. If it does return then ALL registers must be preserved, as it could be interrupting any point in the user s program.

It is not necessary for the software interrupt routine to return if it doesn t want to, it can cause some sort of warm re-start of the user's program.

The software interrupt routine can find out the software interrupt code by reading an EXOS variable CODE_SOFT_IRQ. This is in fact a copy of the code set up by the device since the code itself is reset to zero before jumping to the routine to prevent multiple responses to the software interrupt. If more than one software interrupt occurs before the software interrupt routine can be called then only the most recent one will be acknowledged.

All sources of software interrupts from built in devices can be enabled or disabled by setting appropriate EXOS variables, or making special function calls. The codes from built in devices are:

10h...1Fh	-	?FKEY....	Keyboard function key pressed
20h	-	?STOP	Keyboard STOP key pressed
21h	-	?KEY	Keyboard any key pressed
30h	-	?NET	Network data received
40h	-	?TIME	Timer EXOS variable reached 0

4.3 The STOP key

The stop key is one of the possible sources of software interrupts in EXOS. However it is rather a special case. The reason for this is that pressing the STOP key should always cause an immediate, or almost immediate response. However, the system is frequently waiting in a device driver for something to happen (such as the editor waiting for a key to be pressed), or is just doing something which will take a long time (such as the video driver doing a fill). In these cases if the STOP key only caused a software interrupt there would be no immediate response.

The solution to this is that whenever any device is doing something which is potentially a slow, or non-terminating process, it checks the value of FLAG_SOFT_IRQ periodically. If it contains the code ?STOP then the STOP key has been pressed. The device then immediately, or at least soon, returns back to EXOS with a status code .STOP. Eventually this code will find its way back to the user and the software interrupt will occur.

In fact in some cases the situation is worse than this because it is necessary to interrupt a process which runs with normal EXOS interrupts disabled, so the keyboard is not being scanned. An example of this is the cassette driver writing or reading from tape. However in these cases the device itself contains code to look at the STOP key and will cause both the software interrupt, and the error return itself.

5. SEGMENT ALLOCATION

Segment allocation was explained briefly in the system overview and will be described in more detail here. At cold reset time, EXOS builds up a list of all available RAM segments, testing each one. The system will not function unless at least 32k (two segments) is available, and this must include segment 0FFh which will be the system segment.

The lowest numbered RAM segment is taken out of the list and used as the page zero segment. This segment is never used in any form of allocation, it remains in Z-80 page-0 for evermore.

Each RAM segment in the list can be in one of five different states which are:

Free
Allocated to the user
Allocated to the system
Allocated to a device/extension
Shared between the system and the user

The number of segments in each of these catagories can be determined by making a "return system status" EXOS call (code 20), which is explained in the detailed function call specifications later on.

The system segment (segment 0FFh) is always either allocated to the system or shared, it can never be free. All other segments are initially free except for the page zero segment which is outside this allocation scheme.

5.1 User and Device/Extension Segments

When the user makes an "allocate segment" EXOS call (code 24), if there are any free segments then one of them will be marked as allocated to the user and its segment number will be returned. The user can obtain as many segments as he likes in this way, limited only by the number of segments available. He can also free any segments which he has been allocated by making a "free segment" EXOS call (code 25).

Segments can become allocated to devices/extensions in several ways. A device driver can make an allocate segment call in the same way as the user, and if a segment is available it will be marked as allocated to a device/extension. Also a device can free segments in the same way as the user. Device/extension segments can also become allocated when a system extension is loaded (see details of "load module" EXOS call), or at startup time when an extension ROM is linked in (if the ROM requests one - see section on extension ROM initialisation).

Any segments allocated to devices/extensions or to the user will remain allocated after a warm reset. Also device/extension segments (but not user segments) will remain allocated when a new applications program is started up. Great care must be taken with any device that does allocate RAM segments to itself, to ensure that they are freed when the device has finished with them. Particular care must be taken with device initialisation since a device can be re-initialised and will still have the segments allocated, so it must remember this and not try to allocate itself new segments.

Whenever a user or a device/extension segment is requested, the lowest numbered available segment will be allocated. This ensures that the video segments, which have high numbers, are kept as much as possible for the system so that they will be available for video channels.

5.2 System Segments and the EXOS Boundary

Segments which are allocated to the system are basically used for channel RAM areas. The system uses RAM starting at the top of the system segment, down as far as necessary, possibly continuing into other segments. The top of the system segment is used for system variables, system stack, device RAM areas (see explanation of devices and device descriptors) and RAM areas for extension ROMs. All of these must be contained in the system segment. Below these there is a chain of channel descriptors, each with an associated RAM area, which can occupy as many segments as necessary. This will be described in more detail later.

Any segments which are used for channel RAM are marked as allocated to the system. Each segment is used from the top down until it becomes full, at which time another segment is allocated. Thus all system segments will be fully used, except for the last one which may have some space left in the bottom. There is a system variable the "EXOS boundary" which indicates the lowest address in the last system segment which is being used. This value can be read by doing a "read EXOS boundary" call (code 22) which returns a value in the range [0000h to 3FFFh].

New system segments can be allocated when a channel is opened or when a user device or system extension is linked in. When a channel is closed and the associated channel RAM is freed, this may result in the channel RAM usage moving out of a segment, in which case the segment will be freed and the EXOS boundary set up for the previous segment.

EXOS always allocates the highest numbered segment available when it needs a new segment for the system. This ensures that as much contiguous video RAM as possible is available for video channels, since video segments are the highest numbered. If a video segment becomes free while the system is using a non-video segment then the two will be swapped, although this will only be done next time a channel is opened.

5.3 The Shared Segment and User Boundary

There can be at most one RAM segment which is shared between the user and the system. If it exists, this will always be the last of the segments used by the system and will therefore contain the EXOS boundary as described above.

The user will be allocated a shared segment if he makes an "allocate segment" call when there are no free segments available. This fact is indicated by a specific status code (.SHARE) being returned by the allocate segment call and the user will also be told the current position of the EXOS boundary within this segment (see description of "allocate segment" call). A device or a system extension can never be allocated a shared segment.

When the shared segment is allocated, a second boundary, called the "user boundary", is created within the segment. This is in addition to the EXOS boundary and will initially have the same value. The user can at any time set a new position for the user boundary by making a "set user boundary" call (code 23). The user boundary can be set to any value from zero, up to and including the current setting of the EXOS boundary.

The user can use the segment from the start up to (but not including) the user boundary. EXOS is always using the segment from the top, down to (and including) the EXOS boundary. The area in between the two boundaries (which may be zero bytes) is no man's land and must not be used either by EXOS or by the user. However EXOS may, when it requires more RAM, move the EXOS boundary down as far as the user boundary. Similarly the user may move the user boundary up as far as the EXOS boundary when it needs more RAM. In this way the sharing of the segment between EXOS and the user is flexible and can change.

The segment can become un-shared when a channel is closed, if EXOS no longer needs the segment. Also the user can free the shared segment in which case it will be flagged as allocated to the system. Having freed it, the user can always allocate it again of course.

When a channel is opened, if there is a shared segment then the EXOS boundary will usually have to be moved down. The user boundary should therefore be moved down as far as possible before opening a channel, to make space. Also, if a segment has becomes free while there is a shared segment (it could have been freed by the user or by a device or extension), then EXOS is unable to allocate this to the system, although it can be allocated to the user. This means that it is advisable for the user to free the shared segment as soon as possible, maybe copying the contents into a new segment, in order to make the best use of RAM.

5.4 System Segment Usage

The system segment has been mentioned several times before. This section gives details of how it is used, and certain addresses. Further details of the various sections of RAM which can be allocated in it will be explained in the relevant sections.

The very top of the system segment contains a few variables which are at defined absolute addresses and can be used either by the user or by devices. Some of these have already been explained and others will be mentioned later. This list just gives the address and name of each one, along with a very brief description.

0BFFFh - USR_P3 \
0BFFEh - USR_P2 \ These are the contents of the four paging registers when EXOS was last
0BFFDh - USR_P1 / called. Needed by devices when
0BFFCh - USR_P0 / given user addresses.

0BFFA/Bh - STACK_LIMIT Used for stack checking by devices which need more than the default amount of stack.

0BFF8/9h - RST_ADDR User's warm reset address.

0BFF6/7h - ST_POINTER The Z-80 address of the status line memory. The 42 bytes from this address onwards are the status line (see video driver specification).

0BFF4/5h - LP_POINTER The Z-80 address of the start of the line parameter table (see video driver specification).

0BFF3h - PORTB5 Current value of general output port 0B5h. Used by various devices which access this port. See device driver specs for description.

38 0BFF2h - FLAG_SOFT_IRQ Triggers software interrupts.

36/37 0BFF0/1h - SECOND_COUNTER 16-bit seconds counter.

35 0BFEFh - CRDISP_FLAG Flag for suppressing sign-on message.

Below these fixed variables are all the internal system variables for the EXOS kernel, and also RAM areas for all the built in devices. These RAM areas include space for the line parameter table, character font, function key strings, sound queues, etc, as well as variables for each device. This area also includes space for the EXOS system stack which is used by all devices and system extensions. The size of this area is fixed for any one version of EXOS.

Below this fixed area is the list of RAM segments, and below that the list of extension ROMs, both of which vary in size depending on the number of extension RAM and ROM units connected. Below these lists is any system segment RAM allocated to extension ROMs when they are initialised (see later for explanation). These areas are all set up at cold reset time and then remain fixed.

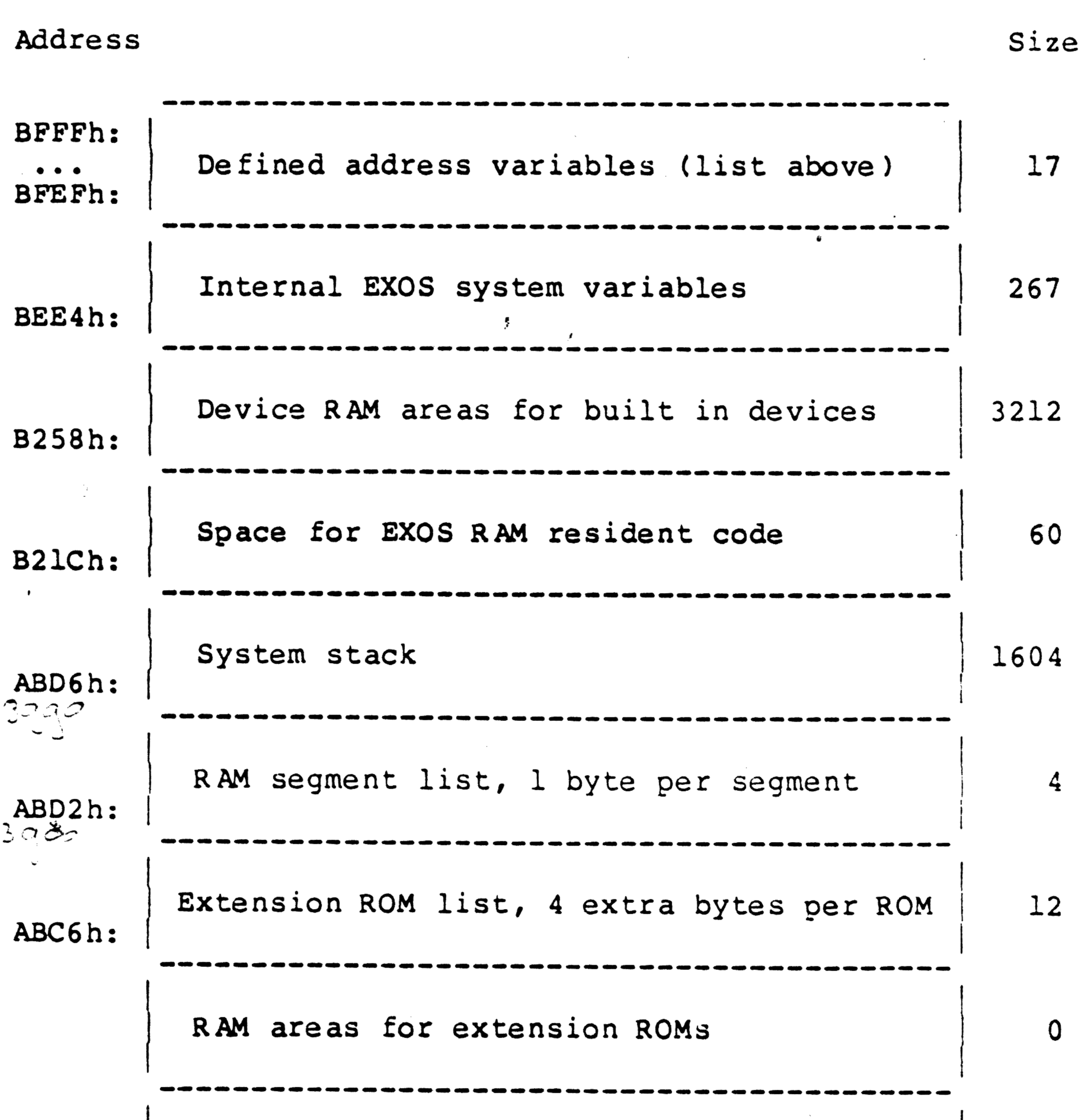
Below this are the device descriptors for all built in device drivers and also any device drivers contained in extension ROMs. This includes any device RAM areas required by extension ROM devices. Built in devices have their device RAM allocated permanently in the fixed RAM area and so do not require any RAM here. This area is newly set up whenever a "reset EXOS" call is made, with the reset flags set to re-link devices (see description of the reset EXOS call), which is generally when a new applications program takes control.

When a user device is linked in, this area will be extended downwards to include any device RAM which the new device requests. This will result in everything below this are being moved down. Once allocated this device RAM will remain until devices are re-linked (see above), which will destroy the user device driver.

All of the above areas must lie wholly within the system segment. Any attempt to allocate RAM which would push them out of this segment will fail.

Immediately below the user device RAM area is the start of the channel RAM area: This must start in the system segment, but can run down into as many other segments as required. The channel RAM area includes a channel descriptor, and a RAM area for each channel which is currently open. These RAM areas can be moved around by EXOS when other channels are opened or closed, or user devices linked in. They are explained in detail in the section on channel RAM allocation.

It is clear from the above description that the sizes and addresses of most of these areas vary depending on the hardware and software configuration. However as an example the diagram below shows the addresses for a standard 64k machine with a single ROM cartridge, such as the IS-BASIC cartridge, fitted. This should only be used as a guide since the exact sizes and addresses may vary in future versions. The addresses are given in Z-80 page-2, since this is where the system segment is normally accessed by EXOS and devices, although it can of course be paged in to any of the Z-80 pages.



6. DEVICE DESCRIPTORS

6.1 The Device Chain

Every device driver has a "device descriptor" in RAM somewhere which defines the device s name, the address of the device driver code and various other details. They are kept in a linked list (called the device chain), and whenever a channel is opened, EXOS searches this list for a device with the correct name and opens the channel to that device.

The device chain is re-built whenever a "reset EXOS" call is made with the reset flags set to re-link devices (see details of the reset EXOS call). This occurs at cold reset time and when a new applications program takes control. The chain is initially created with a descriptor for each of the built in device drivers, and also for any device drivers contained in extension ROMs.

The user, or a system extension, can link in new devices with a simple EXOS call. These will be added to the device chain but will be lost when the chain is re-built.

6.2 Details of Device Descriptors

The format of a device descriptor is given here. Each element is one byte, apart from the device name which is of a variable size. The offsets given are offsets from the DD_TYPE field since this is where the device chain pointers point to.

-3	DD_NEXT_LOW	\	24-bit address of DD_TYPE field of
-2	DD_NEXT_HI	>	next descriptor. Address will be in
-1	DD_NEXT_SEG	/	Z-80 page-1. End of chain indicated by DD_NEXT_SEG=0.
+0	DD_TYPE		Must be zero.
+1	DD_IRQFLAG		Defines device interrupt servicing.
+2	DD_FLAGS		b0 set for video device. b1-b7 clear
+3	DD_TAB_LOW	\	24-bit address of device entry point
+4	DD_TAB_HI	>	table. Address must be in Z-80
+5	DD_TAB_SEG	/	page-1.
+6	DD_UNIT_COUNT		Normally zero. Non-zero to allow multiple devices with this name.
+7..	DD_NAME		Device name string.

The DD_TYPE field is provided to allow for future expansion and also to enable a device to be disabled. This happens for example when a new device is linked in with the same name as an existing one. The old device will be disabled (unless DD_UNIT_COUNT is non-zero - see below).

The DD_IRQFLAG field has one bit for each of the four sources of interrupts in the Enterprise. If the appropriate bit is set then this device driver's interrupt routine will be entered whenever an interrupt of that type occurs. Any combination of bits can be set. The bit assignments are:

```
        b1 - Programmable sound interrupts
        b3 - 1Hz interrupts
        b5 - Video interrrupts (50Hz)
        b7 - External interrupts (network)
b0,2,4,6 - Should be zero.
```

Bit-0 of the DD_FLAGS byte is used to control channel RAM allocation, which is different for video and non-video devices. It will be explained in the section on channel RAM allocation.

The entry point table address (DD_TAB_SEG, DD_TAB_HI and DD_TAB_LOW) points to a table of two byte entry addresses, one for each function which a device has to perform. The address given in the descriptor must be in Z-80 page-1 since EXOS accesses the table there. However the entries in the table itself must be in Z-80 page-3 since when EXOS calls a device it puts the devices code segment in page-3. The entry points themselves must all be in the same segment as the entry point table. The entries in the table are listed here and will be explained in the section on device drivers.

```
+0    Interrupt (Need not be valid if DD_IRQFLAG=0)
+2    OPEN CHANNEL
+4    CREATE CHANNEL
+6    CLOSE CHANNEL
+8    DESTROY CHANNEL
+10   READ CHARACTER
+12   READ BLOCK
+14   WRITE CHARACTER
+16   WRITE BLOCK
+18   READ CHANNEL STATUS
+20   SET CHANNEL STATUS
+22   SPECIAL FUNCTION
+24   Initialisation
+26   Buffer moved
```

The entry points in capitals correspond directly to the relevant EXOS calls, the others are generated inside EXOS.

The DD_UNIT_COUNT field is normally zero but can be set non-zero to allow multiple devices of the same name to be handled by translating unit numbers. This is explained fully in the section on opening channels.

The DD_NAME field is the device name itself. The first byte of this is a length byte, followed by the characters of the name in ASCII. The name can be up to 28 characters long and must consist of upper case letters only.

6.3 Extension ROM Devices

At offset 0008/9h in every extension ROM is a pointer to the start of a chain of devices. If there are no device drivers in the ROM then this pointer should be zero. Each element in the chain is basically a device descriptor as defined above, but with certain fields missing, or replaced by other information. The layout of one of these pseudo-descriptors is:

```
XX_NEXT_LOW     \  16-bit pointer to XX_SIZE field of
XX_NEXT_HI      /  next pseudo-descriptor. In Z-80 page-1
XX_RAM_LOW      \
XX_RAM_HI       /   Amount of device RAM required.

DD_TYPE         \     These fields are exactly as in a
DD_IRQFLAG      |     complete device descriptor defined
DD_FLAGS        |     above.  The DD_TAB_SEG field can
DD_TAB_LOW      \     have any value since EXOS fills
DD_TAB_HI       /     this in when it links the device.
(DD_TAB_SEG)    |
DD_UNIT_COUNT   |
DD_NAME         /

---> XX_SIZE            Size of pseudo-descriptor (see text)
```

The device chain pointer at the start of the ROM points to the XX_SIZE field of the first pseudo-descriptor, in page-1. Similarly the chain pointer (XX_NEXT_LOW and XX_NEXT_HI) in each pseudo-descriptor points to the XX_SIZE field of the next one, in Z-80 page-1. The end of the chain is marked by a pseudo descriptor with both DD_NEXT_HI and DD_NEXT_LOW set to zero.

The XX_SIZE field is a count of the number of bytes in the descriptor from DD_TYPE to the device name. Thus if the device name was one character long, DD_SIZE would be 9.

The main descriptor fields (all those starting with DD_) will simply be copied into RAM when the device is linked in, and a three byte link added to the start to create a complete device descriptor. Note however that EXOS fills in the DD_TAB_SEG field, since a ROM on the expansion stack cannot know what segment it will be in. This means that the entry point table must be in the same segment as the pseudo-descriptor.

The XX_SIZE_HI and XX_SIZE_LOW fields define a 16-bit number which is the amount of device RAM which this device requires in the system segment. This number must be stored in two's complement and with an offset added to allow for the three byte link which EXOS puts on the start of the descriptor. If no device RAM is required then the value should be FFFEh (-2). If one byte is required it should be FFFDh (-3) and so on.

Whenever the device is entered register IY will point to its device descriptor, as will be explained in the section on device drivers. Since the device RAM is allocated immediately below the descriptor, the device RAM can be accessed relative to IY. If "n" bytes are requested then these can be accessed at addresses:

IY-4, IY-5, ..., IY-4-n

6.4 User Devices

User devices are those which are linked in with a "link device" EXOS call which can be made either by the user or by a system extension. To link in a user device a complete device descriptor must be set up in RAM. All fields of this must be complete except for the 24-bit link (DD_NEXT_SEG, DD_NEXT_HI and DD_NEXT_LOW). The EXOS call is then made with DE pointing to the TYPE field of this descriptor, which can be in any Z-80 page.

An area of device RAM can be requested by simply setting register BC to the amount required. This RAM will be allocated below the device RAM for any ROM extension devices. The device driver will be passed the address of this RAM area in register IX when it is first initialised. If "n" bytes are requested then they can be accessed at:

IX-1, IX-2, ..., IX-n

Note that this address will only be passed in IX on the first initialisation. It is the responsibility of the device driver to remember the address for future use, even when it is re-initialised such as after a warm reset.

7. DEVICE DRIVERS

A device driver consists of a set of routines, one to implement each of the fourteen entry points contained in the entry point table which was described in the previous chapter. This chapter describes the functions which must be provided by each of these routines, including details of register usage.

7.1 Device Driver Routines - General

Of the fourteen device driver entry points, eleven of them match up directly with EXOS function codes 1 to 11. Whenever the user makes one of these EXOS calls, EXOS will find out which device is the correct one for this channel and call the appropriate entry point of that device driver. These calls are referred to as the **"device channel calls"**.

The three remaining device driver entry points are, for initialisation, interrupt service and channel buffer moving. Calls to these three routines are originated from within the EXOS kernel at appropriate times and each is discussed in detail below.

Whenever a device driver routine is entered, the segment containing the entry point will be paged into Z-80 page-3. Page-2 will always contain the system segment (segment 0FFh), and page-0 will of course contain the page zero segment. In the case of the device channel calls, the segment containing the channel descriptor and channel RAM (see later) will be in page-1, for other calls the contents of page-1 will be undefined. The stack pointer will be set to the system stack, in Z-80 page-2, and there will be at least 100 bytes available on the stack, in addition to that needed for interrupt servicing (only 50 bytes for an interrupt routine).

When an device driver is called, register IY will always contain the address of the DD_TYPE field of the device descriptor, in Z-80 page-2. In the case of extension devices (linked in from extension ROMs), this can be used to access the device RAM which is allocated immediately below the device descriptor in the system segment. A user device may sometimes have to access RAM relative to its device descriptor, which will not be in the system segment, so it will have to page the correct segment in (remembering to disable interrupts temporarily sine the stack will be paged out). To enable a user device to do this, the segment number of the segment containing its device descriptor is passed in register B' whenever the device is called.

Device driver routines can corrupt all registers, including the index registers and the alternate register set, since they will have been saved by EXOS. The device driver can also corrupt the contents of Z-80 page-1 with impunity, but should exercise caution with the other Z-80 pages. Generally registers A, BC and DE are used to pass parameters to and return results from the routines.

7.2 Device Initialisation Routine

The device initialisation routine is passed no parameters (other than the segment and address of the channel descriptor in B' and IY), and returns no results. It is called when the device is first linked into the system, and again whenever a "reset EXOS" function call is made, which occurs at a warm reset or when a new applications program takes control.

Any channels which the device may have open will vanish when this routine is called, and so any variables or data areas which the device may keep must be reset. Note that any RAM segments allocated to the device will not be freed, so the device must remember that it still has these after subsequent initialisations.

7.3 Channel RAM Allocation

Every channel which is open has an area of "channel RAM" allocated to it. It is the job of the "open channel" or "create channel" routines (described below) to make an "allocate buffer" EXOS call to obtain the required amount of RAM. The allocate buffer EXOS call will be described later. This function call MUST be made before the open or create channel routine returns to EXOS, even if zero bytes of channel RAM are required, since it also sets up a channel descriptor for the channel.

When the "allocate buffer" call is made, it will return the address of the channel RAM in register IX. This will be in Z-80 page-1 and the correct segment will be paged into page-1. Whenever the device driver is entered in future with a channel call to this channel, page-1 and register IX will be set up correctly. If "n" bytes of channel RAM are allocated then they can be accessed at addresses:

IX-1, IX-2, ..., IX-n

The 16 bytes of RAM immediately above the channel RAM (IX+0....IX+15) contain a channel descriptor. This contains system information about the channel and should not be modified by the device.

In the case of non-video devices, the channel RAM will all be in one segment. In the case of video devices however, only a certain amount of the RAM, specified by the device and starting at IX-1, will definitely be in one segment, the rest may carry on down into other segments. If this is the case then each new segment will have a segment number one less than the previous one and they will all be video segments (0FCh to 0FFh). This allows a video device to obtain sufficient RAM for a large video page. Normally only the built in video driver will be a video device, although any device can make itself one simply by having a bit set in its device descriptor (see above).

Once allocated the channel RAM can be moved by EXOS. This can only occur when another channel is opened or closed, or a user device linked in. Since devices are not allowed to make any of these EXOS calls, it is impossible for the channel RAM to be moved while the device driver is executing. Whenever the channel RAM is moved the "buffer moved" entry point of the device driver will be called. This entry point is described below.

7.4 The Buffer Moved Routine

The "buffer moved" entry point is called by EXOS immediately after it has moved a channel buffer of this device. This routine returns no results but is passed the following parameters:

```
b':IY = Device descriptor segment & address (as usual)
   IX = New address of channel descriptor, will be paged
        into Z-80 page-1.
    A = Channel number of channel buffer moved
   BC = Amount that channel buffer has moved
```

The channel buffer may have been moved into a different segment. If the device needs to know this then it can read the new segment number from the page-1 register. The distance moved parameter in register BC is strictly speaking a signed 17-bit number, with the sign bit missing. This means that if, for example, a value of 1 is passed in BC, then this could mean that the buffer has been moved either up by 1 byte, or down by 65535 bytes. In practice this difference does not matter since it only affects the new segment number and this can be determined separately.

Whenever the buffer moved entry point is called, interrupts will be disabled and should not be re-enabled by the device driver. This is to ensure that the device's interrupt routine cannot be called while it is in an intermediate state.

7.5 Device Interrupt Routines

EXOS can handle interrupts from any of the four possible sources on the Enterprise computer (video, sound, 1Hz and external). When an interrupt occurs, EXOS examines the DAVE chip to determine which source it came from. It then scans through the device chain calling the interrupt entry point of any device which has requested servicing of this type of interrrupt (by setting a bit in DD_IRQFLAG in its device descriptor). When all devices have been called, the interrupt is cleared in the DAVE chip, all registers and paging restored and EXOS returns to the interrupted program.

Interrupts are allowed at any time, including while executing device driver code, except while certain system variables are being updated or channel buffers are being moved. Also, interrupts are disabled while servicing an earlier interrupt, so there is no nesting of interrupts. If an interrupt from another source occurs while already servicing an interrupt then it will be held up until servicing of the first one is complete. Thus no interrupts should be missed but they may be serviced late.

The interrupt entry point of a device driver is optional, it is only required if the DD_IRQFLAG field of the device descriptor is non-zero. When a device is linked in, EXOS will ensure that any sources of interrupts which the device wants to service are enabled in the DAVE chip.

The device's interrupt routine will be entered just like any other entry point, with registers B' and IY set up to the device descriptor segment and address as usual. No results are returned from the interrupt routine and all registers can be corrupted (AF, BC, DE, HL, IX, IY, AF', BC', DE', HL'). The entry point will be called with interrupts disabled and they should not be re-enabled, neither should the device attempt to reset the interrupt flag in the DAVE chip - EXOS does that.

There is an EXOS variable (see later) called IRQ_ENABLE_STATE which defines which of the four sources of interrupts are currently enabled. Any of them can be enabled or disabled by changing this EXOS variable and writing it out to the interrupt enable register in the DAVE chip. This should be done with care since the keyboard will not be scanned if video interrupts are disabled so it can be difficult to recover from this.

7.6 Device Channel Calls

The device channel calls are the device entry points which correspond with EXOS function codes 1 to 11. Full details of these EXOS calls can be found in a later section. This section describes them only from the device's point of view.

All of these routines have certain parameters and results in common. These are:

```
Parameters:  B':IY = Device descriptor segment and address
                IX = Pointer to channel RAM in Z-80 page-1.
                 A = Channel number +1 (see next paragraph)
           BC & DE = General parameters to routine

Results:         A = Status code, returned to user
           BC & DE = General results from routine
```

The channel number parameter passed to the device routine is one greater than the channel number as specified by the user. This is due to the way in which EXOS handles channel numbers internally, and means that a device can never be passed a channel number of zero.

The device driver does not need to return with the status register set depending on the value returned in A. The setting of flags is done by EXOS before returning to the user.

7.6.1 Open Channel and Create Channel Routines

For most devices the open channel and create channel routines can be the same. The difference is only relevent for file handling devices, where "open" is intended to open an existing file and "create" is intended to create a new one.

The routine will be passed a pointer to a filename string in DE (length byte first). This will have been copied from the string passed by the user, into a buffer in the system segment, and will have been uppercased and checked for syntax and length (maximum 28 characters). If no filename was specified by the user then this will be a null string.

The unit number specified by the user (or a default) will be passed in register C. Unit numbers are explained in the section on the "open channel" EXOS function.

Assuming that the device decides that it will accept the open channel call, it MUST make an "allocate buffer" call to setup the channel descriptor and obtain any channel RAM which it may need for this channel. Details of this call can be found in the section on EXOS function calls (later). This function call will return a pointer to the RAM in IX and page it into page-1. This is the only case of an EXOS call corrupting any unusual registers or the paging.

7.6.2 Block Read and Write Routines

All devices must provide a block read and a block write routine, which are capable of reading or writing up to 65535 bytes. Some devices (such as disk) will implement these intelligently, doing data transfers directly into the user's buffer. However most devices simply do repeated calls to their own character read or write routines, copying the bytes into or out of the buffer.

Special care must be taken with accessing the user's buffer area. The buffer pointer is passed in DE straight from the user's call. This may point to any address in any of the four Z-80 pages, and refers to the segment which was in that page when the user called EXOS, not when EXOS called the device driver routine. The device driver will therefore have to translate this address to one in Z-80 page-1, and page in the correct segment in order to access the buffer, but must not forget the segment with its channel RAM in. In order to determine the segment number, four variables are provided in the system segment which define the four segments which were paged in when the EXOS call was made. These are called USR_P0, USR_P1, USR_P2 and USR_P3 and their addresses were given in an earlier section. These variables are handled re-entrantly, so they will survive nested EXOS calls correctly.

Note also that the user's buffer can cross a segment boundary and so the segment may need to be changed, and the address adjusted several times. Also the device should cope correctly with a block size of zero bytes, simply returning a zero status code without doing anything.

If an error occurs part way through a block read or write then registers DE and BC should be returned with their values correctly adjusted to indicate how much has been read or written.

8. EXOS VARIABLES

The "read/write/toggle EXOS variable" EXOS call, which will be described later, provides a way for the user, a device driver or a system extension, to access a set of system variables without knowing their actual address. These variables control many aspects of the system, particularly in setting up options for devices before opening channels to them. The ones which are relevant to particular built in device drivers are described in the appropriate device driver specification but a complete list is included here.

Each variable has an 8-bit value, and is identified by an 8-bit EXOS variable number. This list includes all variables which are implemented by the EXOS kernel but there is a facility for system extensions to implement further ones, with numbers above 127 (see next chapter).

Any variable can be set to any value from zero to 255. However many of the variables act as switches to turn something on or off. In these cases, zero corresponds to "on" and 255 to "off". The EXOS call to manipulate them has a "toggle" function which does a ones complement of the value and will thus switch from zero to 255 and vice versa.

0 - IRQ_ENABLE_STATE
b0 - set to enable sound IRQ.
b2 - set to enable 1Hz IRQ.
b4 - set to enable video IRQ.
b6 - set to enable external IRQ.
b1,3,5 & 7 must be zero.

1 - FLAG_SOFT_IRQ. This is the byte set non-zero by a device to cause a software interrupt. It could also be set by the user to cause a software interupt directly. This variable is also available at a fixed address given in an earlier section.

2 - CODE_SOFT_IRQ. This is the copy of the flag set by the device and is the variable that should be inspected by a software interrupt service routine to determine the reason for the interrupt.

3 - DEF_TYPE Type of default device
0 => non file handling device (eg. TAPE)
1 => file handling device (eg. DISK)

4 - DEF_CHAN Default channel number. This channel number will be used whenever a channel call is made with channel number 255.

5	-	TIMER	1Hz down counter. Will cause a software interrupt when it reaches zero and will then stop.
6	-	LOCK_KEY	Current keyboard lock status
7	-	CLICK_KEY	0 => Key click enabled
8	-	STOP_IRQ	0 => STOP key causes soft IRQ <>0 => STOP key returns code
9	-	KEY_IRQ	0 => Any key press causes soft IRQ, as well as returning a code
10	-	RATE_KEY	Keyboard auto-repeat rate in 1/50 second
11	-	DELAY_KEY	Delay 'til auto-repeat starts 0 => no auto-repeat
12	-	TAPE_SND	0 => Tape sound enabled
13	-	WAIT_SND	0 => Sound driver waits when queue full <>0 => returns .SQFUL error .. .. ..
14	-	MUTE_SND	0 => internal speaker active <>0 => internal speaker disabled
15	-	BUF_SND	Sound envelope storage size in 'phases'
16	-	BAUD_SER	Defines serial baud rate
17	-	FORM_SER	Defines serial word format
18	-	ADDR_NET	Network address of this machine
19	-	NET_IRQ	0 => Data received on network will cause a software interrupt
20	-	CHAN_NET	Channel number of network block received
21	-	MACH_NET	Source machine number of network block
22	-	MODE_VID	Video mode \
23	-	COLR_VID	Colour mode \ These variables select the characteristics of
24	-	X_SIZ_VID	X page size / a video page when it
25	-	Y_SIZ_VID	Y page size / is opened
26	-	ST_FLAG	0 => Status line is displayed
27	-	BORD_VID	Border colour of screen
28	-	BIAS_VID	Colour bias for palette colours 8...16
29	-	VID_EDIT	Channel number of video page for editor
30	-	KEY_EDIT	Channel number of keyboard for editor
31	-	BUF_EDIT	Size of edit buffer (in 256 byte pages)
32	-	FLG_EDIT	Flags to control reading from editor

33	-	SP_TAPE	Non-zero to force slow tape saving
34	-	PROTECT	Non-zero to make cassette write out protected file
35	-	LV_TAPE	Controls tape output level
36	-	REM1	\ State of cassette remote controls,
37	-	REM2	/ zero is on, non-zero is off

9. SYSTEM EXTENSION INTERFACE

When EXOS does a cold start it builds up a list of all extension ROMs which are plugged in. Each of these ROMs has a single entry point which is called under various cirumstances with an action code to indicate what function the ROM is to carry out. This chapter describes all the action codes and what the response to them should be.

There is a facility to load programs into RAM and link them in as system extensions. Details of how this is done and the file format will be given in the next chapter. Once loaded these RAM extensions are treated exactly as if they were ROM extensions, and will only be removed when a cold reset is done.

There is an EXOS call provided to pass a string around all system extensions to give them a chance to carry out some function. This results in the extensions being called with action code 2 (command string) or 3 (help string), the meaning of which will be explained in this chapter. Details of the "scan extensions" EXOS call itself will be given in the section on EXOS calls later.

9.1 Calling System Extensions - General

System extensions are called by the EXOS kernel and will always be entered in Z-80 page-3 at their single entry point. Page-2 will contain the system segment (segment 0FFh) which will include the stack, and page-0 will of course contain the page zero segment. ROM extensions can be allocated an area of RAM at cold reset time (see below). The segment containing this RAM will be in page-1, and it will be pointed to by register IY. For RAM resident extensions, page-1 and register IY will be un-defined.

Note that ROM extensions are allowed to make "scan extension" EXOS calls while in their "allocate RAM" routines. This can result in a ROM being entered with action code 2 or 3 before it has had any RAM allocated. This case can be detected by testing for segment number zero in Z-80 page-1, which can only occur before RAM is allocated, or if no RAM is requested.

An action code is always passed in register C, and registers B and DE are used for passing various parameters to, and returning results from, the system extension. All other registers (AF, HL, IX, AF', BC', DE', HL') can be corrupted if desired.

System extensions are normally called by doing an "extension scan", which may originate from a user EXOS call or be generated by the kernel. This involves passing the same action code and parameters to each system extension in turn. If the system extension returns the action code unchanged, then the values passed back in BC and DE will be passed on to the next extension in the list. Thus if a system extension does not support a given action code or command it should return BC and DE unchanged to ensure that the scan continues.

If a system extension returns with register C set to zero then the extension scan will stop, and the values returned in registers BC and DE will be considered as the results - the interpretation depending on the action code. In this case, the value returned in register A is a status code indicating success or failure using the normal EXOS status code values.

The extension scan calls any RAM resident extensions first, followed by any extension ROMs. The very last extension in the chain is the built in word processor program.

9.2 Action Codes

Below are detalis of each of the action codes. Any values not included here are reserved for future extensions and should be ignored by all system extensions, simply returning with BC and DE unchanged. The action codes are described in nùmerical order although the initialisation and ram allocation ones are rather special cases.

A system extension can ignore any of these action codes which it wants to, they are all optional. Any action code which is not supported should be ignored by returning with BC and DE preserved. It is acceptable (although not very useful) for a system extension to consist of just a "RET" instruction at its entry point.

The action codes provided are:

1. Cold reset
2. Command string
3. Help string
4. EXOS variable
5. Explain error code
6. Load module
7. RAM allocation
8. Initialisation

9.2.1 Action code 1 - Cold Reset

This action code is passed around all ROM extensions at cold reset time, when the copyright display program terminates, in order to allow one of them to select itself as the current applications program. The only other time when this action code can be received is when an attempt to load a new applications program fails (see section on "loading functions"). No parameters are passed and no results are returned with this action code.

If the extension wants to set itself up as the current applications program then it simply goes through the normal startup procedure (described below) and does not return from this call. If the extension does not want to do this then it just returns from this call with register C (the action code) preserved.

9.2.2 Action code 2 - Command string

This action code results from a "scan extensions" EXOS call. It is passed a pointer to a string in register DE. This string will have a length byte first and will be stored in a buffer on the stack, so the "scan extensions" call is re-entrant. The first word of the string (up to the first space character) will have been uppercased and register B will contain a count of how many bytes there are in this first word.

The first word is the name of a command, service or program. Each extension will have a set of commands which it recognises. If the extension does not recognise this command then it should return from the call, preserving BC and DE. If it does recognise the command then it should respond to it, possibly interpreting the rest of the string as parameters, returning with register C=0, and a status code in A, unless it wishes other extensions to also respond to this command.

In carrying out the command the system extension can make any EXOS calls required, including further "scan extension" calls. It is often useful to make use of the default channel number for doing screen input/output since it cannot know what other channels are available.

The extension can interpret the command string as a cue to start itself up as the current applications program. For example the strings "BASIC", "LISP" and "FORTH" will be interpreted in this way by the appropriate language cartridges. In order to do this the extension behaves exactly as if it had received action code 1 (cold start). Details of the startup procedure are given below.

9.2.3 Action code 3 - Help string

This action code also results from a "scan extensions" EXOS call, where the first word of the string was "HELP". The "HELP" (and any trailing spaces) will have been removed from the string and then the rest of the string treated exactly as if it was the original string passed to the EXOS call. The parameters for this action code are thus identical to those for action code 2 (command string) described above.

If the string is null (register B will be zero), then this is a general HELP call to all extensions. In this case the extension should just write its name and version to the default channel (using channel number 255) and return with BC and DE preserved.

If the string is not null, and the first word is any of the action code 2 commands recognised by this extension, then specific help information about that command should be printed to the default channel, and register C returned as zero, with a status code in register A (normally zero). If desired the rest of the string can be interpreted as further parameters to control what information is displayed.

If the string is not null and the first word is not a valid command for this extension then registers BC and DE should be returned unchanged.

9.2.4 Action code 4 - EXOS variable

This action code results when a "read/write/toggle EXOS variable" call was made with a variable number not-recognised by the internal ROM (see previous chapter). It allows system extensions to implement additional EXOS variables and may be particularly useful for extension ROMs which also contain extension devices. The parameters passed are:

B = 0, 1 or 2 for READ, WRITE and TOGGLE (ones complement)
E = EXOS variable number
D = New value to be written (only if B=1)

If the variable number is not recognised then the extension should return with BC and DE preserved. If the variable number is one supported by this extension then the appropriate function should be performed and the following parameters returned:

A = status (normally zero)
C = 0
D = New value of EXOS variable

To avoid conflict with the internal EXOS variables, and any others which may be added in future versions or extensions, system extensions should only use EXOS variable numbers of 128 and above.

9.2.5 Action code 5 - Explain error code

This action code results from a user "explain error code" function call. The error code is passed around all system extensions to give them a chance to provide an explanation string. The internal ROM provides explanations for all error codes which can generated by the EXOS kernel or any of the built in devices, unless a system extension returns a string first.

The error code is passed in register B and if it is not recognised the extension should just return with register BC preserved. To avoid conflict with the built in error codes, and any new ones in future versions or extensions, extension ROMs should only use error codes below 7Fh for errors which they generate themselves.

If the error code is recognised then a pointer to an ASCII explanation string (length byte first, maximum length 64 characters) should be returned. This can be in any segment and need not be paged in to the Z-80 memory space when the extension returns. The results returned are:

A - not required, can be any value
B = Segment number containing message
C = 0
DE = Address of message string (can be in any Z-80 page)

9.2.6 Action code 6 - Load module

The details of the Enterprise file module format will be described in the next chapter. This action code is passed around system extensions when a module header of an unrecognised type is read in by EXOS, before returning an error to the user. It allows a system extension to handle loading of its own module types without requiring any special commands.

The extension is passed a pointer to the module header (16 bytes) which will be in the system segment, and also the channel number to load from:

B = Channel number to load from
DE = Pointer to 16 byte module header

The type byte (at DE+1) should be examined to see if this is a module type recognised by this extension. If not then it should return with BC and DE preserved. If the module type is recognised then the rest of the module should be read in from the specified channel, possibly using other parameters from the header, and initialised if this is necessary. Register C should be returned zero, and a status code in A which should be zero if the loading was successful and some error code if not.

9.2.7 Action code 7 - RAM allocation

This action code is rather special since it is only ever called at cold start time, and is only received by ROM extensions. It will only be called once and will always be the first call which the EXOS kernel makes to the ROM. However, as noted above, it is possible for a ROM to be entered with action code 2 or 3 before having any RAM allocated, so if the ROM expects to have RAM it must test for this case by looking for segment zero in Z-80 page-1.

If the ROM does not require any RAM allocation then it should simply ignore this action code, returning register C unchanged. In this case, when future calls are made to this ROM, Z-80 page-1 and register IY will be undefined since there is no RAM area for them to point to.

If the ROM does require RAM to be allocated then it should return the following results:

```
C = 0 (To indicate RAM is required)
B = RAM type flags.   b0 - set for page-2 RAM
                      b1 - set for page-1 RAM
                  b2..b7 - not used. zero.
DE = Number of bytes required
```

The ROM can be allocated one of two types of RAM. Page-2 RAM is allocated in the system segment and so the extension can address it regardless of what it puts in Z-80 page-1. The amount of page-2 RAM is limited since it must all be in one segment and this segment is used for many other purposes. The other type of RAM allocation is page-1 RAM. This is allocated in a segment which the system marks as a device allocated segment, and can be up to very nearly 16k. If this type of allocation is used then more RAM is available, but a whole segment will be taken away from the user. Several extension RAM areas can be put in one segment, and the same segment can also be used for loading the code of relocatable or absolute system extensions into (see next chapter).

The type of RAM allocation required is specified by a pair of flags passed back in register B. If the page-2 flag (bit-0) is set then the RAM will be allocated in the system segment if possible. If the page-1 flag (bit-1) is set then a separate device segment will be used. If both flags are set then the system segment will be used if there is enough space, otherwise a separate device segment will be used.

If the RAM allocation is successful then the address and segment of the RAM area will be saved in the ROM extension list along with the ROM number. Whenever the ROM is called in future the RAM segment will be put in Z-80 page-1 and register IY will point to the RAM area. If the page-1 flag (bit-1 of register B) was clear, so the RAM was allocated in the system segment, then register IY will point to the RAM area in Z-80 page-2. In all other cases IY will point to the RAM in Z-80 page-1, even if the RAM is actually in the system segment (both flags set). If "n" bytes of RAM were requested then they can be accessed at addresses:

IY+0, IY+1, ..., IY+(n-1)

If the RAM allocation failed because there was not enough RAM available then this extension ROM will be marked as invalid in the ROM list and will never be entered again.

Note that the call with this action code is made very early on in the system initialisation, before device drivers have been linked in or initialised. Some EXOS calls are allowed but any of the device related calls (open channel, link device and so on) are not. Generally care should be exercised with EXOS calls made during RAM allocation. As mentioned before, a "scan extensions" call is allowed, and it will scan all ROM extensions, even those which have not yet had RAM allocated. This is the only case in which an extension ROM can be entered before having its RAM allocated - care must be taken with this.

9.2.8 Action code 8 - Initialisation

System extensions are initialised immediately after devices have been initialised. This is done initially at cold reset time (for ROM extensions), and again whenever an "EXOS reset" call whith the appropriate flags set (see later) is made. This occurs when a warm reset happens and also when a new application program takes control. RAM resident extensions are also initialised immediately after they have been loaded. No parameters are passed to the extensions and no results are returned. Register C (the action code) should be preserved but all other registers can be corrupted.

9.3 Starting a New Applications Program

A system extension may decide to start itself up as the current applications program as a result of a call with action code_ 1 or 2. To do this the following procedure should be carried out.

1. Do an "EXOS reset" call with the reset flags set to 60h (see later). This will de-allocate user RAM, abolish any opened channels, re-link and re-initialise all built in and extension devices, abolish any user devices and re-initialise extension ROMs (including the one making the call). It will return with interrupts disabled.

2. Set up a user stack somewhere in the page zero segment, since no other RAM is available, and then re-enable interrupts.

3. Allocate any additional RAM segments which are needed, and open any default channels.

4. Set up a warm reset address for when the reset button is pressed. This should be done even if the program does a complete restart for a warm reset, to ensure that any RAM resident system extensions will remain resident.

5. Set up the default channel number to the program's normal screen I/O channel (usually an editor channel), to allow system extensions to print their help messages.

After doing this, it is in full control as the current applications program and can make any EXOS calls.

10. Enterprise File Format and EXOS Loading Functions

10.1 Enterprise File Format

All files which are to be loaded by EXOS should follow the format described here. It is designed so that the operator of a program such as BASIC can simply give a command such as "LOAD" without knowing what he is going to load. It could be a BASIC internal format program, or it could be a new device driver in relocatable format, to name but two.

A file consits of a series of one or more modules. Each module starts with a 16 byte module header which defines what type of data is to follow in the rest of the module. A file can contain several modules so that, for example a BASIC program can be loaded at the same time as a new device driver which the program uses, simply by having them as two modules in a single file.

The header starts with a null byte (zero) to indicate that it is an Enterprise module header, rather than for example an ASCII text file. Any files which do not start will a null will be referred to as ASCII files although they may be any other sort of data.

Following the null is a type byte, which specifies what type of data the rest of the module contains. The next 13 bytes are different for each type and contain various other parameters such as size and entry point addresses. The very last byte of the header is a version number and should always be zero for current versions.

10.1.1 Module Header Types

The defined types of module are:

0	-	$$ASCII	ASCII File
1	-		Not used
2	-	$$REL	User relocatable module
3	-	$$XBAS	Multiple BASIC program
4	-	$$BAS	Single BASIC program
5	-	$$APP	New applications program
6	-	$$XABS	Absolute system extension
7	-	$$XREL	Relocatable system extension
8	-	$$EDIT	Editor document file
9	-	$$LISP	Lisp memory image file
10	-	$$EOF	End of file module
12...31	-	Reserved for future use by IS/Enterprise	

Type zero is recognised as an ASCII file to reduce the possibility of an ASCII file being mistaken for an Enterprise module header. This will be explained in the section below on the EXOS loading functions.

When a module has been loaded another module may follow, so the system will attempt to load another header. It is therefore necessary to end each file with a module header with the "end of file" type (type 11) to indicate that there is no more to load.

Header types 4, 5, 9 and 10 are specific to particular languages or devices and are described in the documentation for those programs (IS-BASIC, IS-LISP and the EXOS editor). They will not be mentioned further here.

Of the remaining types, numbers 6, 7, and 8 are handled entirely by the EXOS kernel, and type 3 is handled mostly by the kernel but with some interaction by the applications program. All of these types will be described in the following sections.

10.2 Loading Enterprise Format Files

When the user wants to load a file, he should ensure that the channel to load from is open and then make a "load module" EXOS call. This will read one byte from the channel and immediately return a .ASCII error, with the character code in register B, if the byte is non-zero.

If the first byte is zero then another byte (the type byte is read). If this is zero then it is an ASCII file so a .ASCII error is returned, with the type byte (zero) in register B. This ensures that if an ASCII file starts with a series of nulls then it will be recognised as an ASCII file and only the first null will be lost.

If the type byte is non zero then it is saved and another 14 bytes read in to complete the module header. If it is an end of file header (type 11) then a .NOMOD error will be returned. This should be trapped by the user program since it is not really an error, it is the normal terminating condition.

If the module is a type which is handled internally by EXOS (type 6, 7 or 8) then the rest of the module will be loaded in and initialised (details are given in the following sections). If it is not a type handled by EXOS then the module header will be passed around any system extensions to give them a chance to load it if they recognise the type. If the module is loaded in either of these ways then a zero status code will be returned to the user.

Assuming that the module was not loaded by EXOS or by a system extension then a .ITYPE error will be returned to the user, and the module header copied into a buffer passed by the user. The user can then look at the type byte and load the rest of the module if he recognises it.

When a module has been loaded, by the user, by EXOS, or by a system extension, another "load module" call should be made to load in the next module of the file. This will continue until a .NOMOD error is received from EXOS, which is the normal termination, or a fatal error occurs, either from the loading channel or an invalid module, which will result in an error response.

10.3 Relocatable Data Format

EXOS supports the loading of relocatable modules using a simple bit stream relocatable data format. There are two types of relocatable modules, user relocatable modules and relocatable system extensions. These module types and how they are loaded will be described in later sections, this section just describes the relocatable bit stream format itself.

The data of a relocatable module is a bit stream in the sense that individual data fields are a variable number of bits and are not aligned on byte boundaries. The bytes of the data are interpreted most significant bit first, so the first bit of the bit stream is bit-7 of the first byte.

A complete relocatable module consists of a sequence of items which are defined by sequences of bits in the bit stream. The following diagram shows the decoding of the bit stream into the various items. The items themselves are explained afterwards.

```
0          -> 8-bits    load absolute byte
1 00       -> 16-bits   load relocatable word
. 01 0 0   -> 2-bits    set run time page
. .. . 1   ->           restore run time page
. .. 1     -> 16-bits   set new location counter
. 10       ->           end of module
. 11       ->           illegal - for future expansion
```

10.3.1 Location Counter and Run Time Page

When the relocatable loader is called it is passed a starting address which can be in any Z-80 page. It loads the data into whatever segment was in that page, and must not cross a segment boundary. It keeps a location counter which is the current address it is storing bytes at and is also used for loading relocatable words. This location counter is initially set to the start address passed to the loader.

If a "set new location counter" item is found then the following 16 bits form an offset which is added to the current location counter. Adding this offset must not move the location counter into a new page.

It is often useful to have sections of code loaded into a segment which will be accessed in different Z-80 pages, since the segment can be paged into different pages. This is particularly true when creating user device drivers which may be loaded into page-0, but when executed will run in page-3. It is to provide this facility that the "set run time page" and "restore run time page" items are provided.

When a "set run time page" item is found, the following two bits define a new page. The top two bits of the location counter will be set to this new page setting. This will not affect where bytes are actually loaded since the page is irrelevant, as they are always loaded into a single segment. However it will affect the values produced for relocatable words which are loaded. This means that code can be loaded in one page to run in another.

The "restore run time page" item will set the page of the location counter back to what it was when the loader was called, regardless of any new pages which have been set since then.

10.3.2 Relocatable Words and Absolute Bytes

When a "load absolute byte" item is found, the following 8 bits are stored at the current location counter address and the location counter incremented by one. When a "load relocatable word" item is found, the following 16 bits are read and the current location counter added on to them. The resulting word is stored low byte first at the location counter address and the location counter is incremented by two.

10.3.3 End of Module Item

When an "end of module" item is found it will terminate the relocatable loader. Any remaining bits in the last byte will be padded out with zeros and the following byte will be the start of the next module header.

10.4 User Relocatable Modules

User relocatable modules are loaded into user RAM and are regarded as being part of the current applications program once loaded. It is the responsibility of the user to organise allocation of RAM for them to be loaded into. They are useful for providing user device drivers, indeed the interlace video driver which is provided with the Enterprise computer is loaded as a user relocatable module.

The module header for a user relocatable module is:

0	-	zero
1	-	module type (2)
2..3	-	Size of code once loaded
4..5	-	Initialisation offset (0FFFFh if none)
6..15	-	zero

When an EXOS "load module" function call finds a header of this type, it will not recognise it but will just return a .ITYPE error to the user. The user then looks at the type and sees that it is a user relocatable module. The size field in the header defines the complete size of the module once it is loaded. The user must find an area of RAM of this size, in one segment which he can allocate permanently, and pass this address to a "load relocatable module" EXOS call, along with the channel number.

EXOS will load the module into the RAM and then return to the user with a zero status code if there was no error. If the initialisation offset is not 0FFFFh then the user should call this address (the offset is from the initial loading address). This routine will do any initialisation of the module which is required. For example in the case of the interlace video driver, the initialisation will link it into EXOS as a user device.

10.5 Relocatable and Absolute System Extensions

Relocatable and absolute system extensions are loaded automatically by EXOS when the appropriate module header is found. They are loaded into segments which EXOS marks as allocated to devices and will therefore never be freed. Once loaded they function exactly like ROM based system extensions, with a single entry point which is passed action codes. Operation of the extensions once loaded was described in a previous chapter, this section just covers the actual loading and header format.

EXOS maintains a list of segments allocated in this way. They can be used for loading relocatable and absolute extensions, and also for allocating RAM to ROM extensions at cold start time. Absolute extensions always go at the bottom of a segment and so there can only be one per segment. Relocatable extensions and RAM areas for ROM extensions are allocated from the top of a segment downwards and there can be as many of these in a segment as will fit.

The module header format for the two types is the same except for the type byte:

```
    0  -  zero
    1  -  module type (6 for absolute, 7 for relocatable)
  2..3 -  Size of code once loaded  (< 16k)
 4..15 -  zero
```

EXOS will first allocate enough RAM to load the extension into, which may require allocation of a new segment or may be able to make use of a space in an earlier segment. The data will then be loaded into the segment. In the case of an absolute extension the data will be loaded with the first byte going at address 0C00Ah, which will be the entry point of the extension. For relocatable extensions the code will be loaded anywhere in the segment (addressed in Z-80 page-3) and the entry point will be the very first byte loaded.

If an error occurs in loading then the extension will be lost and the RAM for it will be de-allocated which may involve freeing a segment if it was a newly allocated one. If no error occurs then the new extension will be linked on to the start of the list of system extensions and then initialised, as described in the chapter on system extensions. Control will then return to the user in the usual way.

10.6 New Applications Programs

The "new applications program" module type is loaded automatically by EXOS when the header is found. It can be used to load programs of up to 47.75k. The program it loads will automatically be started up as the new applications program, losing the previous one. It is intended for loading programs such as machine code games from cassette although it will have other uses.

The module header format is:

```
    0  -  zero
    1  -  module type (5)
  2..3 -  Size of program in bytes (low byte first)
 4..15 -  zero
```

EXOS will look at the size of the program and work out if enough user RAM can be allocated to load it into, allowing for a shared segment but without closing any channels. If there is not enough then a .NORAM error is returned, otherwise EXOS will commit itself to loading the file.

Having reached this stage it will allocate the necessary user RAM segments for the program and from this point on it cannot return to the current applications program since it will have corrupted the RAM it was using. If an error occurs from here on then it will display an error message on the default channel and then scan all extensions with the cold start action code. This is the only time that extensions can receive a cold start action cold other than at a genuine cold start.

Once the required segments have been allocated the new program will be read in from the channel and stored as absolute bytes starting at address 100h. When the whole program has been loaded, EXOS will simulate a warm reset to the start of the program at 100h. This warm reset will be done with the reset flags set to 20h (see later) which will completely reset the I/O system, without disturbing user RAM. The new applications program will have to go through the normal startup procedure (described earlier), except that it needn't do another EXOS call.

Since user segments may have had to be allocated to load the program in, the program may be occupying a shared segment. If this is the case then the user boundary will have been set to just above the end of the program to allow as much RAM as possible for opening channels etc.

11. EXOS Function Calls in Detail

This chapter contains details of all the EXOS function calls. Many of them have been described earlier in general terms. This section concentrates on details such as register usage and error codes, and describes the function calls from the point of view of the program making the call.

Parameters are passed to EXOS calls in registers A, BC and DE, and results are passed back in the same registers. Register A returns a status code which is zero if the call was successful and a non-zero error code otherwise. All other registers (HL, IX, IY, AF', BC', DE', HL') are preserved by all EXOS calls, and also the user's paging is not disturbed. EXOS calls can be made from any address in any Z-80 page, and the user's stack can be in any of the four pages.

11.1 Device Name and Filename String Syntax

The "open channel" and "create channel" function calls take a string parameter. This string defines which device driver the channel is being opened to, and also specifies a unit number and filename. The syntax of the string is:

[[device-name] [["-"] unit-number] ":"] [file-name]

where [] denotes an optional part and "" delimits literal characters.

The device name can be up to 28 characters and must be entirely letters, which will be uppercased before using so case is not significant. If it is not present then EXOS will use a default device name which can be set with a "default device name" EXOS call (code 19). If the unit number is also absent (see below) then the default unit number, which can also be set with this call, will be used.

The unit-number, if present, can be seperated from the device name with a single "-" (minus) character if desired or it can immediately follow it. The unit number consists of a series of decimal digits which will be converted into a one byte value by EXOS. If the device name is specified with no unit number, then a default unit number of zero is used.

The optional filename consits of up to 28 characters which can include letters, digits and the special characters "\/-_." (not including the quotes). Letters will be uppercased before the string is used. If there is no filename then it will just be taken as the null string.

The filename and unit number will be passed through to the device driver for interpretation. However if the device driver has the DD_UNIT_COUNT field in its device descriptor set then some manipulation of the unit number will occur.

If the DD_UNIT_COUNT field is set to "N" then this means that the device driver only accepts unit numbers in the range [0 ... N-1]. If the unit number is greater than this then it will be reduced by "N" and the search of the device chain will continue. When another device of the same name is found the process will be repeated and if it is now within range then the device will be called with the reduced unit number. In this way several devices with the same name can be supported, with the distincion being by unit number. This is not used by any built in devices but could be used by add on disk units.

11.2 Function 0 - System Reset

Parameters: C = Reset type flags
Results: A = Status (always zero but flags ot set
Interrupts disabled

This call causes a reset of EXOS. The flags passed in register C control exactly what the RESET does, as below.

b0 ... b3 must be zero

b4 - Set => Forcibly de-allocate all channel RAM, and re-initialise all devices. User devices will be retained.

b5 - Set => As bit-4 but also re-link in all built in and extension devices, and re-initialise system extensions. User devices will be lost. Device segments are not de-allocated.

b6 - Set => De-allocate all user RAM segments.

b7 - Set => Cold reset. This is equivalent to switching the machine off and on again. All RAM data is lost.

Note that the status register is not set to be consistent with the status code (which is always zero anyway) and registers BC', DE' and HL' are corrupted by this EXOS call. Also a side effect of the call is that interrupts are disabled.

An automatic RESET call (with flags set to 20h) is done when a warm reset occurs. Also a RESET (with flags set to 60h) must be done by a system extension when it takes control as a new current applications program.

11.3 Function 1 - Open channel

Parameters: A channel number (must not be 255)
DE pointer to device/filename string
Results: A status

The format of the filename string was specified above. The filename and unit number are passed to the device driver for interpretation and many devices will just ignore them. If the device is one which supports filenames then it will return an error code if the file specified does not already exist. Some devices require options to be selected (by special function calls) before the channel can be used. Also some devices require parameters to be specified by setting EXOS variables before a channel can be opened.

The unit number is ignored by all built in devices except the network driver. If a device name with no unit number is specified then a default of zero is used which devices could translate into their own internal default if desired.

For the open channel function to be successfully completed, the device must allocate itself a channel buffer before it returns and an error may be returned if there is insufficient RAM available.

11.4 Function 2 - Create channel

Parameters: A channel number (must not be 255)
DE pointer to device/filename string
Results: A status

The create function is identical to the open function except that if the device supports filenames, then the file will be created if it doesn't exist, and an error code returned if it does. It is identical to OPEN CHANNEL for all built in devices except the cassette driver.

11.5 Function 3 - Close channel

Parameters: A channel number (must not be 255)
Results: A status

The close function flushes any buffers and de-allocates any RAM used by the channel. Further reference to this channel number will result in an error. The device's entry point is called before the channel RAM is de-allocated.

11.6 Function 4 - Destroy channel

Parameters: A channel number (must not be 255)
Results: A status

The destroy function is identical to the close function except that on a file handling device the file is deleted. It is identical for all built in devices.

11.7 Function 5 - Read character

Parameters: A channel number
Results: A status
B character

The read character call allows single characters to be read from a channel without the explicit use of a buffer. If no character is ready then it waits until one is ready. This call is passed directly through to the device driver.

11.8 Function 6 - Read block

Parameters: A channel number
BC byte count
DE buffer address
Results: A status
BC bytes left to read
DE modified buffer address

The read block function reads a variable sized block from a channel. The block may be from 0 to 65535 bytes in length and can cross segment boundaries. Note that the byte count returned in BC is valid even if the status code is negative, although not if it is an error such as non-existent channel. This allows a partially successful block write to be re-tried from the first character which failed. This call is passed directly through to the device driver.

11.9 Function 7 - Write character

Parameters: A channel number
B character
Results: A status

The write character function allows single characters to be written to a channel. This call is passed directly to the device driver.

11.10 Function 8 - Write block

Parameters:
- A channel number
- BC byte count
- DE buffer address

Results:
- A status
- BC bytes left to write
- DE modified buffer address

The block write function allows a variable sized block to be written to a channel and is similar to block read. The byte count returned in BC is valid even if the status code is negative. This call is passed directly through to the device driver

11.11 Function 9 - Channel read status

Parameters:
- A channel number

Results:
- A status
- C 00h if character is ready to be read
  FFh if at end of file
  01h otherwise.

The read channel status function call is used to allow polling of a device such as the keyboard without making the system wait until a character is ready. This call is passed directly through to the device driver.

11.12 Function 10 - Set and Read Channel Status

Parameters:
- A channel number
- C Write flags
- DE pointer to parameter block (16 bytes)

Results:
- A status
- C Read flags

This function is used to provide random access facilities and file protection on file devices such as disk or a RAM driver. The format of the parameter block is:

bytes:
- 0...3 - File pointer value (32 bits)
- 4...7 - File size (32 bits)
- 8 - Protection byte (yet to be defined)
- 9...15 - Zero. (reserved for future expansion)

The assignment of bits in the read and write flags byte is as below. The specified action is taken if the bit is set.

	WRITE FLAGS	READ FLAGS
b0	Set new pointer value	File pointer is valid
b1	not used (0)	File size is valid
b2	Set new protection byte	Protection byte is valid
b3...b7	not used (0)	always 0

This allows the file pointer and/or the protection byte to be set independently, or just to be read. Not all devices need to support this function, indeed none of the built in devices support it. If a device doesn't support it then it should return a .NOFN error code.

11.13 Function 11 - Special function

Parameters:
A channel number
B sub-function number
C unspecified parameter
DE unspecified parameter

Results:
A status
C unspecified parameter
DE unspecified parameter

This function call allows device specific functions to be performed on a channel. If it is not supported by a device then a .ISPEC error will be returned.

The sub-function number specified in register B determines which special function is required. Sub-function numbers should be different for all devices, unless equivalent functions are implemented. The special functions for built in devices are (see device driver specifications for details):

@@DISP = 1	VIDEO - Display page
@@SIZE = 2	VIDEO - Return page size and mode
@@ADDR = 3	VIDEO - Return video page address
@@FONT = 4	VIDEO - Reset character font
@@FKEY = 8	KEYBOARD - Program function key
@@JOY = 9	KEYBOARD - Read joysick directly
@@FLSH = 16	NETWORK - Flush output buffer
@@CLR = 17	NETWORK - Clear input and output buffers
@@MARG = 24	EDITOR - Set margins
@@CHLD = 25	EDITOR - Load a document
@@CHSV = 26	EDITOR - Save a document

All other sub-function codes from zero to 63 are reserved for use by IS/Enterprise. Codes of 64 and above can be used by user devices.

11.14 Function 16 - Read, Write or Toggle EXOS Variable

Parameters: B = 0 To read value
= 1 To write value
= 2 To toggle value
C = EXOS variable number (0...255)
D = New value to be written (only for writ
Results: A = Status
D = New value of EXOS variable

This function allows EXOS variables to be set or inspected. These variables control various functions of the system and specific devices. Note that the value is returned in D even for write and toggle. A list of currently defined EXOS variables was given earlier. System extensions can implement additional EXOS variables.

11.15 Function 17 - Capture channel

Parameters: A - Main channel number
C - Secondary channel number (0FFh to cancel capture)
Results: A - Status

The capture channel function causes subsequent read function calls (read character, read block and read status) to the main channel, to read data instead from the secondary channel. When the function call is made, the main channel must exist but no check is made on the secondary channel number existing.

The capture applies to all subsequent input from the main channel number until either the secondary channel is closed or gives any error (such as end of file) or the main channel is captured from somewhere else. The effect of the capture can be cancelled by giving a secondary channel number of 0FFh which is not a valid channel number.

11.16 Function 18 - Re-direct channel

Parameters: A - Main channel number
C - Secondary channel number (0FFh to cancel redirection)
Results: A - Status

The re-direct function causes subsequent output sent to the main channel with write character or write block function calls, to be sent to the secondary channel instead. The redirection lasts until the secondary channel is closed or returns an error, or the main channel is redirected somewhere else. A secondary channel number of 0FFh will cancel any redirection of the main channel.

11.17 Function 19 - Set default device name

Parameters: DE - device name pointer (no colon)
C - device type 0 = non file handling
1 = file handling
Results: A - status

The set default device name function specifies a device name and (optionally) a unit number which will be used in subsequent "open channel" or "create channel" function calls if no device name is specified by the user. Initially the default name will be "TAPE-1" but will be set to "DISK-1" if a disk device is linked in. The specified device name and unit number are checked for legality (ie. no invalid characters) but not for existence in the device chain.

If a string with only a unit number, such as "45" is specified then this will set a new unit number but the default name will be un-changed. If device name but no unit number is given, then the default unit number will be set to zero.

The "device type" given in register C is simply copied to the "device type" EXOS variable. This will be zero in the default machine because the default device is "TAPE" which is not a file handling device. If a disk unit is connected then the device type will be set to 1. This variable is not currently used by EXOS but can be of some use to applications programs.

11.18 Function 20 - Return system status

```
Parameters:    DE -> Parameter block, 8 bytes.
Results:        A =  Status code, always 0.
                B =  Version number (currently 20h)
               DE - unchanged
```

This function returns the version number of the system and various parameters which describe the RAM segment usage in the system. The parameters returned are, in order:

0. Shared segment number (0 if no shared segment)
1. Number of free segments.
2. Number of segments allocated to user, excluding page-zero segment and shared segment (if there is one).
3. Number of segments allocated to devices.
4. Number of segments allocated to the system, including the shared segment (if there is one).
5. Total number of working RAM segments.
6. Total number of non-working RAM segments.
7. *** Not currently used ***

11.19 Function 21 - Link Device

```
Parameters:    DE - Pointer to RAM in Z-80 space
                    containing device descriptor.
               BC - Amount of device RAM required.
Results:        A - status
```

The link device function causes the device descriptor pointed to by DE to be linked into the descriptor chain. The descriptor will be put at the start of the chain and any existing device with the same name will be disabled. DE must point at the TYPE field of the descriptor and the descriptor must not cross a segment boundary. Once linked in the user must ensure that the device code and descriptor are not corrupted until a RESET function call with bit-5 set (to un-link user devices) has been made.

The amount of RAM requested will be allocated in the system segment. When the device is first initialised, this RAM area will be pointed to by IX and the device must remember this address since it will never be told it again, even when it is re-initialised.

11.20 Function 22 - Read EXOS Boundary

07B1 - 2017

Parameters: none
Results:
A - status (Always zero)
C - Shared segment number. 0 if there is no shared segment.
DE - EXOS boundary in shared segment (0..3FFFh)

The read EXOS boundary function returns the offset within the currently shared segment, of the lowest byte which the system is using. If there is no shared segment then DE will point to where the EXOS boundary would be if a shared segment were allocated.

11.21 Function 23 - Set User Boundary

Parameters: DE - Offset of new USER boundary. (0...3FFFh)
Results: A - Status

The set user boundary function allows the user to move the USER boundary within the currently shared segment. If there is no shared segment then this function is not allowed. The boundary may not be set higher than the current EXOS boundary.

11.22 Function 24 - Allocate Segment

Parameters: none
Results:
A - status
C - Segment number
DE - EXOS boundary within segment

The allocate segment function allows the user to obtain another 16K segment for his use. If a free segment is available then it will be allocated and status returned zero with segment number in C and DE will be 4000h.

If there are no free segments but the user can be allocated a shared segment, then the segment number will be returned in C and DE will be the initial EXOS boundary. In this case a .SHARE error will be returned. The user boundary is initially set equal to the EXOS boundary.

If there are no free segments and there is already a shared segment then a .NOSEG error will be returned.

If this function call is made by a device driver then the segment will be marked as allocated to a device and a shared segment cannot be allocated.

61

11.23 Function 25 - Free segment

Parameters: C - Segment number
Results: A - status

The free segment function allows the user to free a 16k segment of RAM. The segment must be currently allocated to the user or be shared. The page zero segment cannot be freed as it was never allocated explicitly with an "allocate segment" call.

If this function call is made by a device driver then it must be to free a segment which was allocated to a device driver with an "allocate segment" call. There is no checking of which device is freeing the segment - devices are supposed to be well behaved.

11.24 Function 26 - Scan System Extensions

Parameters: DE = Pointer to command string
Results: A = Status

This function causes the string to be passed around all system extensions after some processing, with action code 2 (or 3 if the first word of the string is "HELP"). This allows services to be carried out by system extensions and also allows transfer to a new applications program.

11.25 Function 27 - Allocate Channel Buffer

Parameters: DE - Amount of buffer which must be in one segment
BC - Amount of buffer which needn't be in one segment (only needed for video devices)
Results: A - status
IX -> Points newly allocated buffer
PAGE-1 contains the new buffer segment

The allocate channel buffer function is provided only for devices and may not be called by the applications program. It is used to provide a channel with a RAM buffer when it is opened. The "multi segment size" passed in register BC is ignored for non-video devices since they must have their channel buffer all in one segment. So for non-video devices BC need not be loaded before making the call.

11.26 Function 28 - Explain Error Code

Parameters: A - Error code which needs explaining
DE - Pointer to string buffer (64 bytes)
Results: A = 0
DE - Unchanged

This function allows an EXOS error code to be converted into a short text message. System extensions are given a chance of doing the translation. All error codes generated by the EXOS kernel and the built in devices are explained by the internal ROM. If the string returned is of zero length then it is an error code which no one was willing to explain.

11.27 Function 29 - Load Module

Parameters: DE -> Buffer for module header (16 bytes)
B = Channel number to load from
Results: A - Status
DE = Unchanged
B - If A=.ASCII - 1st character of file
If A=.ITYPE - Module type
Else un-defined

This function call was explained in the section on loading Enterprise module format files. It will load a module header and then either load the module itself, or pass it to the system extensions for loading. If the system extensions don't want it then it will be returned to the user in his buffer (pointed to by DE), for him to load.

If a module is loaded OK by EXOS or a system extension then a zero status code is returned. In this case, or if the module is successfully loaded by the user, the "load module" function call should be repeated to load the next module. This should continue until a .NOMOD error is returned which indicates that an "end if file header" was read, or until a fatal error occurs.

If the first byte is not zero, or the type byte is zero then the file is not an Enterprise format file and a .ASCII error is returned with the first character in B. The user can then do what he wants with the ASCII data, but should not attempt to load another module from this file.

11.28 Function 30 - Load Relocatable Module

Parameters:	B =	Channel number to load from
	DE =	Starting address to load at
Results:	A =	Status
	DE =	Unchanged

This function call can be used by the user to load user relocatable modules, with header type 2, which will be rejected by the "load module" call above.

The user must find the correct sized chunk of RAM to load the module into (from the size in the header). If the function call returns a zero error code then the user should call the initialisation entry point of the code loaded (if there is one) and should then call "load module" again to get the next module header. This is explained in more detail in an earlier chapter.

11.29 Function 31 - Set Time

Parameters:	C =	Hours	0...23 (BCD)
	D =	Minutes	0...59 (BCD)
	E =	Seconds	0...59 (BCD)
Results:	A =	Status	

This function sets the internal system clock. The parameters are checked for legality and a .ITIME error returned if they are illegal.

11.30 Function 32 - Read Time

Parameters:	none		
Results	A =	Status	
	C =	Hours	0...23 (BCD)
	D =	Minutes	0...59 (BCD)
	E =	Seconds	0...59 (BCD)

This function reads the current value of the system clock. This clock is incremented every second, using the Enterprise's 1Hz interrupt. When it reaches midnight the date will automatically be incremented (see below).

11.31 Function 33 - Set Date

Parameters:	C = Year	0...99 (BCD)
	D = Month	1...12 (BCD)
	E = Day	1...31 (BCD)
Results:	A = Status	

This function sets the internal system date. The parameters are checked fully for legality, including the number of days in each month and leap years. The year is origined at 1980 so a year value of 4 actually represents 1984. This allows the date to go well into the future (obsolescence built out !).

11.32 Function 34 - Read Date

Parameters:	none	
Results:	A = Status	
	C = Year	0...99 (BCD)
	D = Month	1...12 (BCD)
	E = Day	1...31 (BCD)

This function reads the current value of the internal system calender. This can be set by the user and will increment automatically when the system clock reaches midnight, coping correctly with the number of days in each month including leap years.

++++++++++ END OF DOCUMENT ++++++++++