ENTERPRISE TECHNICAL INFORMATION

****

EXOS 2.0
November 1984

CONTENTS
****

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

Contents

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.

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.

```
         +----+----+----+----+----+----+----+----+
  00h    |        Reserved for CP/M emulation     |
         +----+----+----+----+----+----+----+----+
  08h    | Free                                   |
         +----+----+----+----+----+----+----+----+
  10h    | Free                                   |
         +----+----+----+----+----+----+----+----+
  18h    | Free                                   |
         +----+----+----+----+----+----+----+----+
  20h    | Free                                   |
         +----+----+----+----+----+----+----+----+
  28h    | Free                                   |
         +----+----+----+----+----+----+----+----+
  30h    |       EXOS system call entry vector    |
         +----+----+----+----+----+----+----+----+
  38h    | Interrupt vector   | Soft ISR ad. |    |
         +----+----+----+----+----+----+----+    +
  40h    |                                        |
         +                                        |
  48h    |        Reserved for EXOS code/data     |
         +                                        +
  50h    |                                        |
         +                   +----+----+----+----+
  58h    |                   |                    |
         +----+----+----+----+                    +
  60h    |                                        |
         +        Reserved for CP/M emulation     +
  68h    |                                        |
         +             (Default FCB)              +
  70h    |                                        |
         +                                        +
  78h    |                                        |
         +----+----+----+----+----+----+----+----+
  80h    |                                        |
   .     .        Reserved for CP/M emulation     .
   .     .                                        .
   .     .          (Default buffer area)         .
  F8h    |                                        |
         +----+----+----+----+----+----+----+----+
```

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.

13

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

0BFF2h	-	FLAG_SOFT_IRQ	Triggers software interrupts.
0BFF0/1h	-	SECOND_COUNTER	16-bit seconds counter.
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.

Address		Size
BFFFh: ... BFEFh:	Defined address variables (list above)	17
BEE4h:	Internal EXOS system variables	267
B258h:	Device RAM areas for built in devices	3212
B21Ch:	Space for EXOS RAM resident code	60
ABD6h:	System stack	1604
ABD2h:	RAM segment list, 1 byte per segment	4
ABC6h:	Extension ROM list, 4 extra bytes per ROM	12
	RAM areas for extension ROMs	0
AB42h:	Device descriptors for built in devices	132
AB41h: . .	Start of channel descriptor chain	

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 apects 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 \ These variables select
23	-	COLR_VID	Colour mode \ 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 numerical 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
```

41

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.

43

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 not 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

07E1 - 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 ++++++++++


```
;*************************************************************************
;
;
;   This file defines EXOS variables, function codes, error codes and software
; interrupt codes as externals.  It should be assembled to a .REL file and
; then linked with any code which uses EXOS.  It produces no object code, just
; defines symbols.
;
;
err_code         defl    100h
;
err             MACRO    name
err_code         defl    err_code - 1
                 PUBLIC  name
name             equ     err_code
                ENDM
;
;
;
var     MACRO name,number
name    equ     number
        PUBLIC  name
        ENDM
;
;
;
;-------------------------------------------------------------------------------
;
;                         EXOS FIXED VARIABLES
;                       ========================
;
;
        var     USR_P3,          0BFFFh  ;Four segments which were in Z-80 space
        var     USR_P2,          0BFFEh  ; when EXOS was last called.
        var     USR_P1,          0BFFDh
        var     USR_P0,          0BFFCh
;
        var     STACK_LIMIT,     0BFFAh  ;Bottom limit of stack for devices.
;
        var     RST_ADDR,        0BFF8h  ;Warm reset address
;
        var     ST_POINTER,      0BFF6h  ;Address of status line RAM
        var     LP_POINTER,      0BFF4h  ;Address of start of video LPT.
;
        var     PORTB5,          0BFF3h  ;Current contents of Z-80 port 0B5h.
;
        var     FLAG_SOFT_IRQ,   0BFF2h  ;Flag <>0 to cause software interrupt.
;
;
;
;-------------------------------------------------------------------------------
```

```
;                         FUNCTION CODES
;                         ================
;
;
        var     @RESET, 0               ; Reset system
        var     @OPEN,  1               ; Open channel
        var     @CREAT, 2               ; Create channel
        var     @CLOSE, 3               ; Close channel
        var     @DEST,  4               ; Destroy channel
        var     @RDCH,  5               ; Read character
        var     @RDBLK, 6               ; Read block
        var     @WRCH,  7               ; Write character
        var     @WRBLK, 8               ; Write block
        var     @RSTAT, 9               ; Read status
        var     @SSTAT, 10              ; Set channel status
        var     @SFUNC, 11              ; Special vartion
        var     @EVAR,  16              ; Set/read/toggle EXOS variable
        var     @CAPT,  17              ; Capture channel
        var     @REDIR, 18              ; Re-direct channel
        var     @DDEV,  19              ; Set default device
        var     @SYSS,  20              ; Return system status
        var     @LINK,  21              ; Link device
        var     @READB, 22              ; Read EXOS boundary
        var     @SETB,  23              ; Set USER boundary
        var     @ALLOC, 24              ; Allocate segment
        var     @FREE,  25              ; Free segment
        var     @ROMS,  26              ; Locate ROMs
        var     @BUFF,  27              ; Allocate channel buffer
        var     @ERRMSG,28              ; Return error message.
```

```
;***********************************************************************
;
;
;                      ERROR CODES
;                      ===========
;
;
;       General errors returned by the EXOS kernel
;
        err     .IFUNC  ;0FFh   Invalid function code
        err     .ILLFN  ;0FEh   EXOS function call not allowed
        err     .INAME  ;0FDh   Invalid name string
        err     .STACK  ;0FCh   Insufficient stack
;
        err     .ICHAN  ;0FBh   Invalid channel number (0FFh) or channel
                        ;           does not exist.
        err     .NODEV  ;0FAh   Device does not exist  (OPEN/CREATE)
        err     .CHANX  ;0F9h   Channel already exists (OPEN/CREATE)
        err     .NOBUF  ;0F8h   No ALLOCATE BUFFER call made (OPEN/CREATE)
        err     .BADAL  ;0F7h   Bad ALLOCATE BUFFER params. (OPEN/CREATE)
        err     .NORAM  ;0F6h   Insufficient RAM for buffer.
        err     .NOVID  ;0F5h   Insufficient video RAM.
;
        err     .NOSEG  ;0F4h   No free segments (ALLOCATE SEG)
        err     .ISEG   ;0F3h   Invalid segment (FREE SEG, SET BOUNDARY)
        err     .IBOUN  ;0F2h   Invalid user boundary (SET USER BOUND)
        err     .IVAR   ;0F1h   Invalid EXOS variable number
        err     .IDESC  ;0F0h   Invalid device descriptor type (LINK DEV)
;
;
;
;       General errors returned by various devices
;
        err     .ISPEC  ;0EFh   Invalid special function code
        err     .2NDCH  ;0EEh   Attempt to open second channel
        err     .NOFN   ;0EDh   Function not supported
        err     .ESC    ;0ECh   Invalid escape character
        err     .STOP   ;0EBh   Stop key pressed
;
        public  .SWI
;
;
;
;       File related errors
;
        err     .NOFIL  ;0EAh   File does not exist
        err     .EXFIL  ;0E9h   File exists (CREATE)
        err     .FOPEN  ;0E8h   File already open
        err     .EOF    ;0E7h   End of file met in read
        err     .FSIZE  ;0E6h   File is too big
        err     .FPTR   ;0E5h   Invalid file pointer value.
        err     .PROT   ;0E4h   Protection violation
;
;
;
```


69

```
;       Keyboard errors
;
err     .KFKEY  ;0E3h   Invalid function key number
err     .KFSPC  ;0E2h   Run out of function key space
;
;
;
;       Sound errors
;
err     .SENV   ;0E1h   Envelope is too big or number 255.
err     .SENBF  ;0E0h   Not enough room to define envelope
err     .SENLO  ;0DFh   Envelope storage requested too small (ie. <2)
err     .SQFUL  ;0DEh   Sound queue is full (and WAIT_SND <> 0)
;
;
;
;       Video errors
;
err     .VROW   ;0DDh   Invalid row number to scroll
err     .VCURS  ;0DCh   Attempt to move cursor off page
err     .VCOLR  ;0DBh   Invalid colour passed to INK or PAPER
;
err     .VSIZE  ;0DAh   Invalid X or Y size to OPEN
err     .VMODE  ;0D9h   Invalid video mode to OPEN
err     .VDISP  ;0D8h   Naff parameter to DISPLAY
err     .VDSP2  ;0D7h   Not enough rows in page to DISPLAY
;
err     .VBEAM  ;0D6h   Attept to move beam off page
err     .VLSTY  ;0D5h   Line style too big
err     .VLMOD  ;0D4h   Line mode too big
err     .VCHAR  ;0D3h   Can't display character on graphics page
;
;
;
;       Serial errors
;
err     .BAUD   ;0D2h   Invalid baud rate
;
;
;
;       Editor errors
;
err     .EVID   ;0D1h   Invalid video page for OPEN.
err     .EKEY   ;0D0h   Trouble in communicating with keyboard.
err     .ECURS  ;0CFh   Invalid co-ordinates for positioning cursor.
;
;
;
;       Cassette errors
;
err     .CCRC   ;0CEh   CRC error from cassette driver
;
```

```
;
;       Network errors
;
        err     .SEROP  ;0CDH   Serial device open - cannot use network
        err     .NOADR  ;0CCH   ADDR_NET not set up
;
;
;
;
printe  macro   e
          .printx       * Smallest error code is e *
        endm
;
;
        IF1
          printe        %err_code
        ENDIF
;
;
;------------------------------------------------------------------------------
;
;                 WARNING CODES
;                 ================
;
        var     .SHARE, 07Fh            ;Shared segment allocated
```


71

```
;***************************************************************************
;
;
;                         EXOS VARIABLE NUMBERS
;                         =====================
;
;
;
;
        var     $IRQ_ENABLE,    0       ; Interrupt enable bits.
;
        var     $FLAG_SIRQ,     1       ; Flag to cause a software interrupt.
        var     $CODE_SIRQ,     2       ; Software Interrupt code.
;
        var     $DEF_TYPE,      3       ; Type of default device.
        var     $DEF_CHAN,      4       ; Default channel number
;
        var     $LOCK_KEY,      5       ; Keyboard lock status.
        var     $CLICK_KEY,     6       ; Key click enable/disable.
        var     $STOP_IRQ,      7       ; Software interrupt on STOP key.
        var     $KEY_IRQ,       8       ; Software interrupt on any key press.
        var     $RATE_KEY,      9       ; Keyboard auto-repeat rate.
        var     $DELAY_KEY,     10      ; Delay before auto-repeat starts.
;
        var     $TAPE_SND,      11      ; Tape sound enable/disable.
;
        var     $WAIT_SND,      12      ; Sound driver waiting if buffer full
        var     $MUTE_SND,      13      ; Sound mute enable/disable.
        var     $BUF_SND,       14      ; Sound envelope storage size.
;
        var     $BAUD_SER,      15      ; Serial baud rate.
        var     $FORM_SER,      16      ; Serial word format.
        var     $ADDR_NET       17      ; Network address of this machine
        var     $NET_IRQ,       18      ; Software interrupt on network.
        var     $CHAN_NET,      19      ; Channel of network interrupt
;
        var     $MODE_VID,      20      ; Video mode.
        var     $COLR_VID,      21      ; Video colour mode.
        var     $X_SIZ_VID,     22      ; Video X page size.
        var     $Y_SIZ_VID,     23      ; Video Y page size.
;
        var     $ST_FLAG,       24      ; Status line displayed flag.
        var     $BORD_VID,      25      ; Border colour.
        var     $BIAS_VID,      26      ; Fixed bias colour.
;
        var     $VID_EDIT,      27      ; Video channel number.
        var     $KEY_EDIT,      28      ; Keyboard channel number.
        var     $BUF_EDIT,      29      ; Size of edit buffer.
        var     $FLG_EDIT,      30      ; Editor control flags
;
        var     $SP_TAPE,       31      ; Cassette I/O speed.
        var     $PROTECT,       32      ; Cassette protection control
        var     $REM1,          33      ; Cassette remote 1
        var     $REM2,          34      ; Cassette remote 2
```

```
;*************************************************************************
;
;
;               SOFTWARE INTERRUPT CODES
;               ==========================
;
;
        var     ?FKEY,   10h             ;Function keys 10h...1Fh
        var     ?STOP,   20h             ;Stop key
        var     ?KEY,    21h             ;'any key' 21h
;
        var     ?NET,    30h             ;Network data received.
;
;
;
;
```


73

ENTERPRISE PIN OUT INFORMATION
****

Enterprise pin outs as follows:-
All looking into connector on Enterprise - Pin B1 top left
Pin A1 bottom left

Edge fingers are on 2.54mm pitch, some positions are not used but still counted.

Control 1/Control 2
A1 - Keyboard J (Common)
A2 - Keyboard L
A4 - KB4(9) (Right)
A5 - KB2(7) (Down)
A6 - KB0(5) (Fire)
B1 - OV
B4 - +5V
B5 - KB3(8) (Left)
B6 - KB1(6) (Up)

Numbers in brackets are control 2. All signals are TTL levels - read as part of keyboard matrix. Use multi-core screened cable.

Serial/Network
A1 - Reference
A3 - RTS
A4 - CTS
B1 - OV
B3 - Data Out
B4 - Data In

Use 6 core screened cable

Signal levels relative to
OV line 0 = OV
1 = +12V
Relative to ref. line
0 = -5V
1 = +7V

For networking connect 'RTS' to 'CTS' to form 'control bus', and 'data in' to 'data out' to form 'data bus'. Reference is an offset 'ground', this may not be possible with certain equipment confirgurations.

Printer
A1 OV
A2 /strobe
A3 Data 3
A5 Data 2
A6 Data 1
A7 Data 0
B1 OV
B2 /ready
B3 Data 4
5 Data 5
B6 Data 6
B7 Data 7

Use 12 way flexible ribbon cable.
All signals at TTL levels.

Monitor
A1 Green signal
A2 OV
A3 Monochrome composite video
A4 /HSYNC
A5 /VSYNC
A7 Left audio
B2 OV
B3 Blue signal
B4 Red signal
B5 /CSYNC
B6 Mode switch (Peritel)
B7 Right audio

Use multi core screened cable. All syncs are TTL levels R,G and B levels are 0 to 4 volts linear (not TTL).

1. INTRODUCTION

The cassette driver allows storage of data files on cassette tape. Two cassette recorders can be handled, with separate remote control of the motors on each, allowing reading from one and writing to the other. The files stored can contain any data, not just ASCII.

The files are stored on tape in "chunks" with each chunk being up to 4k bytes. Data is always written to or read from the tape in complete chunks, with the motor being stopped between chunks. However these chunks are buffered within the cassette driver so the user can read or write in any sized blocks or single characters.

Two data rates are provided for recording, these are approximately 1000 and 2400 baud. When a file is read in the speed is determined from the leader automatically.

The cassette driver makes use of the status line for displaying messages when it is loading and saving, and also to display the cassette loading meter. This loading meter can be used to set the level optimally when reading tapes.

2. CASSETTE FILE FORMAT

The details of how data is stored on cassette will be covered later. This section just describes the general format of a file on tape as it appears to the user.

A file consists of a "header chunk", followed by one or more "data chunks". Each chunk will be preceded by necessary information for syncronising the tape reading routine and establishing the speed, including a long leader tone. Details of this are given later.

2.1 The Header Chunk

The header chunk does not contain any data from the file. It contains the filename (which may be null), and a protection flag which can be used to prevent simple tape to tape copying (see later). The header chunk is used to identify the file.

2.2 Data Chunks

Each data chunk contains exactly 4k of data from the file, except for the last one which may have any amount from zero bytes up to 4k. This smaller chunk is used to mark the end of the file. The data within each chunk is split up into 256 byte blocks, with a CRC check done on each block. This ensures that a bad chunk will be rejected fairly quickly.

3. USING THE CASSETTE DRIVER

The cassette driver allows a maximum of two channels to be open to it at a time, one for reading and one for writing. A channel which is opened for reading cannot be written to and vice versa. A reading channel is opened by making an "open channel" call and a writing channel by making a "create channel" call. The cassette driver is thus the only built in device driver which distinguishes between these two EXOS calls.

3.1 Opening Channels for Reading

A channel can be opened for reading a file from tape by simply doing an "open channel" EXOS call with device name "TAPE:". Any unit number is ignored, the filename is optional. The cassette channel will require a channel RAM buffer of 4k (enough for one data chunk) and an error (.NORAM) will be returned if there is insufficient RAM. An error will also be returned if a cassette read channel is already open (.2NDCH), or if there is a protection violation (.PROT - see below).

Assuming that all this was OK, the cassette driver will start the cassette motor (see section on remotes below), and start searching for a suitable header chunk. At this stage it will display the message "SEARCHING" on the status line.

When a header chunk is found, the name of this file from the header is examined. If it is not the same as the filename specified by the user, and if the user's filename was not null, then this is the wrong file. In this case the message "FOUND <filename>" will be displayed on the status line and the search for a suitable header chunk will continue.

If the filename is correct, or if the user's filename was null which means "load the first file found", then the message "LOADING <filename>" will be displayed on the status line and the "open channel" routine will return with a zero status code to indicate success, after stopping the cassette motor. Read character or read block function calls can now be made to read the data.

At any point in this process, the STOP key can be pressed which will abort the searching and return immediately to the user. An internal flag will be set so that any attempt to read characters will result in an .EOF error. The user must close the channel.

The "LOADING" message is left on the status line until the channel is closed. It may of course be overwritten by the user.

3.2 Reading Data

Data can be read from a cassette read channel by simply making any combination of EXOS read character and read block function calls. Data from the tape is buffered in the 4k channel RAM area. When the channel is first opened this buffer will be marked as empty.

If there is data in the buffer when a read character call is made then the next byte will just be returned to the user immediately, and the buffer pointers adjusted.

If there is no data in the buffer when a read character call is made then another data chunk must be read from the tape. The tape motor will be started and the cassette driver will look for a chunk. When a chunk is found, if it is a header chunk then a .CCRC error is returned to the user and the end of file flag will be set so that no more bytes can be read. If a data chunk is found then the data from it will be read into the buffer with CRC checking.

Having read the chunk in successfully, the first character will be returned to the user. If this was the last chunk in the file then a flag is set which will prevent another chunk from being loaded. When this final chunk has all been read by the user, any further read character calls will result in .EOF errors being returned.

If a CRC error occurred in one of the 256 byte blocks in the chunk, then any previous blocks will be buffered as usual and can be read by the user. When all the valid blocks have been read, the next read character call will return a .CCRC error, and subsequent ones will return .EOF errors. No more data can be read from this channel.

The STOP key is tested all the time while data is being read from the tape. If it is pressed then it will cause an immediate return to the user. The end of file flag will be set so that any further read character calls will result in .EOF errors being returned. No data from the interrupted chunk, or any later chunks, can be read.

3.3 Creating Channels for Writing

A "create channel" EXOS call will be accepted as long as there is 4k of channel RAM available for the data buffer, a cassette write channel is not already open and there is no protection violation (see below). This is the same as for a reading channel.

Assuming that this is OK, the cassette driver will start the cassette motor and wait for a second or so for the tape speed to stabilise. The message "SAVING <filename>" will be displayed on the status line. After the delay the cassette driver will write out a header chunk for this new file, which will contain the filename, and then stop the motor. After this it will return to the user with a zero status code to indicate that the create was successful.

The STOP key is tested during the writing of the header chunk and if it is pressed then the write will stop immediately and the channel will be marked as invalid so no data can be written to it. The channel will still be open and so must be closed by the user.

The "SAVING" message on the status line is left there until the channel is closed, although it may be overwritten by the user of course.

3.4 Writing Data

Data can be written to a cassette write channel by any combination of write character and write block calls. A write block call is treated exactly as if each character in the block was written individually. Data is written into a 4k buffer in channel RAM and is only written out to the tape when the buffer becomes full, or the channel is closed.

When a character is written, if is just added to the buffer and the buffer pointers adjusted. If there is still more room in the buffer then the cassette driver will return to the user immediately. If the buffer is now full then it will be written to tape as a data chunk.

The process of writing the data chunk is very similar to writing out the header chunk when the file was created. The motor is started and then there is a delay to allow it to come up to speed. The data chunk itself is then written out and the motor stopped. .This process is interruptable with the STOP key.

3.5 Closing Channels

The cassette driver treats the "close channel" and "destroy channel" EXOS calls identically. When a write channel is closed then the final data chunk must be written out. This is done even if there are no bytes in the buffer, to mark the end of the file. If the STOP key was pressed while data was being written then the channel will have been marked as dead, and in this case the final block will not be written out.

For any cassette channel the status line will be blanked to remove the "LOADING" or "SAVING" message from the status line.

4. MISCELLANEOUS CASSETTE FEATURES

4.1 The STOP Key

The STOP key is tested whenever the cassette driver is actually accessing the tape, either for reading or for writing. Since the cassette driver disables normal EXOS interrupts while it is accessing the tape, it does not rely on the normal keyboard driver detection of the STOP key. Instead it tests for the stop key directly itself and simulates the keyboard's action. However it does not test the STOP_IRQ EXOS variable, so the STOP key will always halt cassette operations even if STOP_IRQ is non-zero.

When the STOP key is detected, a .STOP error will be returned to the user, and also a software interrupt will be caused with software interrupt code ?STOP. The channel which was being used will be marked as no longer valid so that the cassette driver will reject further read or write character calls.

4.2 Tape Output Speed and Level

The data rate for tapes being read is determined automatically from the leader signal, and the level has to be set by the user. However the speed and level for writing out of data are controlled by EXOS variables which must be set before opening the channel, unless the defaults are required

LV_TAPE determines the approximate peak to peak output level of the cassette driver as follows:

```
0 or 1  =>  20 mV
     2  =>  40 mV    (default)
     3  =>  80 mV
     4  => 170 mV
     5  => 350 mV
6...255 => 700 mV
```

SP_TAPE selects between two tape output speeds, a fast speed and a slow speed as follows: (The baud rates are approximate because the actual rate depends on the data since a one and a zero bit take different amounts of time.)

```
SP_TAPE=0  =>  Fast speed  (approx. 2400 baud - default)
SP_TAPE<>0 =>  Slow speed  (approx. 1000 baud)
```

4.3 Cassette Loading Level Meter

The cassette loading level meter is displayed whenever the cassette driver is searching for or reading a chunk from the tape. It derived directly from a hardware level detection circuit, separate from the cassette input circuit, and is displayed on the status line as either a red or a green block next to each other.

If the input level is increased it will become red and if the level is reduced it will become green. The optimum level is when it is just on the verge of changing between red and green, and it may in fact flash between the two as data comes in. Although this is the optimum level, the cassette input is not very sensitive to levels and a wide margin around this optimum level is acceptable.

The level meter is removed from the status line when a chunk has been read, to indicate that the tape is no longer being accessed.

The changing of the level meter is done by changing palette colours 2 and 3 of the status line. When the level meter is removed, these colours are restored to their original values. However in the meantime, if there is anything else on the status line it may change colour.

4.4 Remote Control Relays

The Enterprise is equipped with two remote control relays which enable it to control two tape recorders separately. The motor is started when the cassette driver wants to read or write a chunk and stopped as soon as this has been completed, or the STOP key pressed.

If only one cassette channel is open, then both relays will be activated so either socket can be used. This ensures that reading and writing can be done without moving the remote plug. If both a read channel and a write channel are open, then the read channel will use the remote next to the CASSETTE IN socket (remote 1) and the write channel will use the one next to the CASSETTE OUT socket (remote 2).

The remote relays can also be controlled by two EXOS variables (called REM1 and REM2), separately from the cassette driver. When one of these is changed then the appropriate relay will be set on or off as appropriate. The cassette driver always updates these variables when it uses the remotes so the variables always represent the current state of the relays. Like all other on/off EXOS variables, zero corresponds to "on" (motor going) and 0FFh corresponds to "off" (motor stopped).

The remote relays will always be set off when a reset I/O system occurs (see EXOS kernel specification). This occurs at a warm reset and when a new applications program takes control.

4.5 Use Without Remote Control

Cassette recorders without remote control can be used, provided the PAUSE button on the recorder is used at the correct times. For simply loading and saving programs, no pausing is necessary, assuming that the program doing the loading and saving is fast enough (IS-BASIC is).

For saving, pausing is only necessary to avoid long gaps between chunks, it is not essential. For loading, pausing is necessary to ensure that the data chunks are not missed while the machine is processing the data.

To help with this, when the cassette driver has finished reading a data chunk, and there are more data chunks to follow, it displays a "PAUSE" message on the status line in place of the level meter. This message will remain until the next chunk is required, when it will be overwritten with the level meter again.

4.6 Cassette Sound Feedthrough

The EXOS variable TAPE_SND is used to control feedthrough of the tape input signal to the main sound output. If it is 0FFh then there is no feedthrough. If it is zero then the tape input signal will be fed to the hi-fi sound output and the internal speaker (if MUTE_SND is zero), but not to the headphone/cassette output (to avoid feedback problems). The default setting is on (zero).

4.7 Copying Protection

Since two cassette channels are supported, one for reading and one for writing, it is very easy to open appropriate channels and copy any file at all, regardless of its content, onto another tape. This can be done fairly easily with a BASIC "COPY" command. The cassette driver contains a facility for protecting a file against this very simple type of copying.

When a file is created, and the header written out, the current value of the EXOS variable PROTECT is copied into the header. If this is zero (the default) then the file is not protected. IF it is non-zero then the file is marked as a protected file.

When a read channel is opened, and the header is read in, the "protect" flag from the header is examined. If it is zero then no special action is taken. If it is non-zero then the cassette driver will not permit a write channel to be open at the same time as this read channel. Thus if a write channel is already open then this "open channel" call will be rejected with a .PROT error. If this "open channel" is accepted then further "create channel" calls will be rejected with the same error.

5. HARDWARE

The hardware will not be explained in any detail but the various ports and bit assignments are covered here.

The cassette data input, level detection input, remote control outputs and sound feed-through control are all available as bits on I/O ports as follows:

data input	port 0B6h	bit 7
level input	port 0B6h	bit 6
remote 1 output	port 0B5h	bit 6
remote 2 output	port 0B5h	bit 7
feedthrough toggle	port 0B5h	bit 5

The cassette output is in fact the same as the sound output, with the cassette out socket doubling as a headphone socket. The cassette output is therefore done by using the DAVE chip in D/A mode and so several of the DAVE chip registers are used. These will not be detailed here as they are defined in the DAVE chip specification.

6. CASSETTE DATA FORMAT

As mentioned before a file consists of a series of chunks, the first of which is a header chunk and the rest of which are data chunks. The section describes the format of a chunk in some detail. A header chunk is in fact a special case of a data chunk with a special block count as will be explained later.

6.1 Cassette Signals

Each byte is stored on tape as a series of 8 bits, high bit first, with no start or stop bits. Each bit is stored as a single cycle, with a different frequency to indicate whether the bit is set or clear. These frequencies are in a ratio of 2:3 which is large enough to allow the software to distinguish them when reading in, but small enough to keep the data rate high.

An intermediate frequency is used for the leader tone which comes before the data, and this leader is used to determine the data rate when reading.

A single cycle of lower frequency is used to indicate the start of the data and also establish the phase of the signal (since it may or may not be inverted).

When the data is being read back, all timing is done in terms of whole cycles rather than half cycles. This ensures that it is relatively insensitive to changes in duty cycle which can result from level changes (drop-outs etc.).

The actual whole cycle times and frequencies used for the two tape speeds are:

	Fast Speed	Slow Speed
Leader cycle	424us (2358 Hz)	1000us (1000 Hz)
One bit	344us (2907 Hz)	800us (1250 Hz)
Zero bit	504us (1984 Hz)	1200us (883 Hz)
Sync bit	696us (1437 Hz)	1600us (625 Hz)

6.2 Overall Chunk Format

Each chunk starts with a synchronisation sequence. This consists of several seconds of leader frequency to allow the cassette recorder amplifiers and automatic recording level circuits to stabilise, and to establish the data rate. This is followed by a single low frequency sync cycle to establish the phase of the signal, and then one unused byte to recover from this one long pulse. The next byte always has the value 06Ah and is to ensure that false synchronisation does not occur.

After this byte is the data of the chunk, followed by a few cycles of leader frequency (the trailer) to ensure a clean end to the data. The overall format is shown in this diagram:

```
 -------- ------ ------- ------ --..........-- ---------
| Leader | Sync | Dummy | 06Ah |     Data     | Trailer |
 -------- ------ ------- ------ --..........-- ---------
```

6.3 Internal Chunk Format

The data within a chunk is split up into a maximum of sixteen 256 byte blocks, each starting with a byte count and ending with a two byte CRC check. These blocks are preceded in the chunk by a single, one byte, block count which defines how many blocks there are in the chunk.

The format of the data within a chunk is therefore as shown in this diagram:

```
 ------------- --------- --------- ...... ---------
| Block count |  BLOCK  |  BLOCK  |      |  BLOCK  |
 ------------- --------- --------- ...... ---------
              /           \
     --------              ------------------
    /                                        \
    ----------- --...-- --------- ----------
   | Byte count |  Data  | Low CRC | High CRC |
    ----------- --...-- --------- ----------
```

A header chunk has a block count of 255, but always contains exactly one data block (of varying size). A data chunk has a block count of 0 to 16, and each block is always 256 bytes long, except for the last block in the last chunk of the file. A block containing 256 bytes of data has a byte count of zero, which cannot be misinterpreted because zero is not a valid quantity.

6.4 CRC Checking

Each data block ends with a 16 bit CRC check. This is calculated by treating all bytes of the data block (not including the byte-count byte) as a bit stream. A 16 bit CRC register is initialised to zero at the start of the block and the following process carried out on each bit:

1) XOR the new bit into b15 of the CRC register
2) If the new b15 is set then XOR the CRC with 0810h
3) Rotate the CRC one bit left, putting b15 into b0

Note that this is the same CRC algorithm as that used by the network driver.

7. QUICK REFERENCE SUMMARY

7.1 EXOS Calls

OPEN CHANNEL - Opens a read channel and gets the header chunk. Only one read channel allowed. Device name "TAPE:", unit number ignored. Filename compared with file on tape (unless null). No EXOS variables need be set up before open.

CREATE CHANNEL - Opens a write channel and writes the header chunk. Only one write channel allowed. Device name "TAPE:", unit number ignored. Filename written into header. EXOS variables LV_TAPE, SP_TAPE and PROTECT must be set up before create.

CLOSE/DESTROY CHANNEL - Treated identically. For a write channel will write out any buffered data.

READ CHARACTER/BLOCK - Only allowed for read channel. Read characters from buffer until empty, then read another data chunk from tape.

WRITE CHARACTER/BLOCK - Only allowed for write channel. Writes characters into buffer and writes it out to tape when it gets full.

READ STATUS - Always returns C=0.

SET STATUS - Not supported.

SPECIAL FUNCTION - No special functions

7.2 EXOS Variables

```
LV_TAPE  -  Tape output level (1...6)
SP_TAPE  -  Tape output speed.  0=fast, 0FFh=slow
PROTECT  -  Non-zero to write out protected file

TAPE_SND -  Non-zero to suppress tape sound feed-through
REM1     -  \ Control tape remote relays.
REM2     -  /       0 for ON,  0FFh for OFF.
```

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

1. Overview of Features

The keyboard driver only allows one channel to be open to it at a time. A channel can be opened by giving the device name "KEYBOARD:", any filename or unit number being ignored. If there is already a channel open to the keyboard driver then an error (.2NDCH) will be returned. No EXOS variables need to be set up before opening the channel.

The keyboard device has an interrupt routine which scans the keyboard matrix every video frame (50 times per second). This detects key presses, translates them into ASCII codes, and buffers a single character.

It supports programming of the eight function keys. Each one can be programmed separately for shifted and unshifted use giving effectively sixteen function keys. If the string programmed into any one of these function keys is of zero length then instead of returning characters, this function key will cause a software interrupt when it is pressed.

The eight function keys also each return a specific code if used with the CTRL or ALT keys, giving effectively another 16 functions.

The keyboard driver treats the joystick as if it were four cursor keys and provides diagonal movement by alternating two cursor codes. Autorepeat is supported on all keys. Both the delay until autorepeat begins, and the autorepeat rate can be altered.

The keyboard provides audible feedback by triggering the sound device to produce a click whenever a key is pressed. This can be disabled by the user.

2. Character Input

All input is done using the EXOS read character and read block calls. Read block is supported for compatibility with other devices although it is not very likely to be used. The keyboard is not an output device and so will not accept write character or write block function calls.

With the exception of the function keys which can be programmed with arbitrary strings, each key produces a single ASCII code. Many keys will produce different codes when used in conjunction with the CTRL, SHIFT or ALT keys.

2.1 Lock Modes

The keyboard is always in one of four modes: Normal, shift-lock, caps-lock or alt-lock. The default mode is normal. The mode can be changed by various key combinations:

CTRL LOCK	-	Enters Caps-lock mode.
SHIFT LOCK	-	Enters Shift-lock mode.
ALT LOCK	-	Enters Alt-lock mode.
LOCK	-	Returns to Normal mode.

When the keyboard is in any of the lock modes then it behaves as if the appropriate SHIFT, CTRL or ALT key was held down permanently. If the appropriate key is held down during a lock mode then it temporarily counteracts the effect of the lock. Thus for example in SHIFT LOCK mode the action of the SHIFT key is effectively reversed. In this example if the CTRL key is used while in SHIFT LOCK mode it will behave as if it was in NORMAL mode. This applies to all other combinations.

The current lock mode is indicated on the status line the first six characters of which are reserved for the keyboard. It displays the word SHIFT, CAPS or ALT as appropriate and is blank for normal mode.

There is an EXOS variable (LOCK_KEY) which is always set to the current lock status according to the following codes:

0	-	Un-locked
1	-	CAPS lock
2	-	SHIFT lock
8	-	ALT lock

If this EXOS variable is changed by the user then the next keyboard interrupt will update the lock mode appropriately. Any values other than the above which are put into the variable will be changed to one of the four allowed values.

2.2 Key Codes

These are the ASCII codes returned by each key both normally, and with SHIFT, CTRL and ALT. (All values are in hexadecimal.)

Key	NORMAL	SHIFT	CONTROL	ALT
1	31	21	31	31
2	32	22	32	32
3	33	23	33	33
4	34	24	34	34
5	35	25	35	35
6	36	26	36	36
7	37	27	37	37
8	38	28	38	38
9	39	29	39	39
0	30	5F	1F	9F
-	2D	3D	2D	2D
^	5E	7E	1E	9E
@	40	60	00	80
[	5B	7B	1B	9B
;	3B	2B	3B	3B
:	3A	2A	3A	3A
]	5D	7D	1D	9D
\	5C	7C	1C	9C
,	2C	3C	2C	2C
.	2E	3E	2E	2E
/	2F	3F	2F	2F
space	20	20	20	20
A	61	41	01	81
B	62	42	02	82
C	63	43	03	83
D	64	44	04	84
E	65	45	05	85
F	66	46	06	86
G	67	47	07	87
H	68	48	08	88
I	69	49	09	89
J	6A	4A	0A	8A
K	6B	4B	0B	8B
L	6C	4C	0C	8C
M	6D	4D	0D	8D
N	6E	4E	0E	8E
O	6F	4F	0F	8F
P	70	50	10	90
Q	71	51	11	91
R	72	52	12	92
S	73	53	13	93
T	74	54	14	94
U	75	55	15	95
V	76	56	16	96
W	77	57	17	97
X	78	58	18	98

Y	79	59	19	99
Z	7A	5A	1A	9A
ENTER	0D	0D	0D	0D
ESC	1B	1B	1B	1B
TAB	09	09	09	09
DEL	A0	A1	A2	A3
ERASE	A4	A5	A6	A7
INS	A8	A9	AA	AB
STOP	03	03	03	03
joy up	B0	B1	B2	B3
joy down	B4	B5	B6	B7
joy left	B8	B9	BA	BB
joy right	BC	BD	BE	BF
Function 1	.	.	F0	F8
Function 2	.	.	F1	F9
Function 3	.	.	F2	FA
Function 4	.	.	F3	FB
Function 5	.	.	F4	FC
Function 6	.	.	F5	FD
Function 7	.	.	F6	FE
Function 8	.	.	F7	FF

3. Special Features

3.1 Keyclick control

When the interrupt routine detects a key press it calls a routine called KEYCLICK in the sound driver which produces an audible click. This routine is only called if the EXOS variable KEY_CLICK is zero. Thus setting this variable to a non-zero value will disable key click.

3.2 Autorepeat Control

Autorepeat is controlled by two EXOS variables. DELAY_KEY which is the delay until autorepeat starts and RATE_KEY which is the delay between each repetition of the key. Both of these are in units of 1/50 seconds. DELAY_KEY should always be longer than RATE_KEY and if DELAY_KEY is zero then autorepeat is disabled.

3.3 Function Key Programming

There are sixteen logical programmable function keys numbered 0 to 15. Keys 0 to 7 refer to the basic function keys, 8 to 15 are the shifted versions. Any one of these may be programmed with a string of characters (which may include control codes etc.) using a special function call. The default string for all keys is a null string.

```
Parameters:   B = @@FKEY (=8) (Special function code)
              C = Function key number (0..15)
             DE = Pointer to string (Length byte first)
Returns:      A = Status
```

The maximum length for each programmed string is 23 characters excluding the length byte. An error (.KFSPC) will be returned if the string is too long.

If the programmed string is of zero length (null string) then this function key will cause a software interrupt when it is pressed. The software interrupt code will be ?FKEY (10h) for function key 0, up to ?FKEY+15 (1Fh) for function key 15.

3.4 Stop Key Control

There is an EXOS variable called STOP_IRQ which controls the action of the STOP key. If it is non-zero then the stop key simply returns the ASCII Ctrl-C code (03h) in the same way as all other keys. If STOP_IRQ is zero then instead of this a software interrupt is caused, with software interrupt code ?STOP (20h).

3.5 Hold Key Control

When the HOLD key is pressed the keyboard driver hangs up in its interrupt routine until the HOLD key is pressed again. This will thus freeze any listing etc. which is being produced. When it hangs up it calls a routine in the sound driver to silence the DAVE chip since any sounds will be frozen.

When the HOLD key is pressed it displays the message "HOLD" in place of the current lock mode on the status line. This message will be replaced by the correct lock mode message (which is blank for normal mode) when the hold is released. If the STOP key is pressed while in hold mode then this will force an exit from hold mode, and will then respond to the STOP key in the normal way.

While in hold mode the internal EXOS clock will still be updated so it will not loose time.

3.6 Normal Key Software Interrupts

When a normal key is pressed the character code for it is simply put in the buffer. However if the EXOS variable KEY_IRQ is non-zero then as well as returning the character code, a software interrupt will be caused with software interrupt code ?KEY (21h).

3.7 Direct Joystick Reading

A special function call is provided which will directly read the joystick on the main keyboard, or one of the two external joysticks on the control ports. The parameters for this are:

```
Parameters:     B = @@JOY (=9) (Special function code)
                C = 0  (internal joystick)
                  = 1  (external joystick 1)
                  = 2  (external joystick 2)
Returns:        A = Status
                C = b0  -  Set if RIGHT pressed
                    b1  -  Set if LEFT pressed
                    b2  -  Set if DOWN pressed
                    b3  -  Set if UP pressed
                    b4  -  Set if FIRE pressed
                 b5..b6 -  Clear
```

Note that for the internal joystick the "fire" button is in fact the space bar.

4. Quick Reference Summary

4.1 EXOS calls.

OPEN/CREATE CHANNEL - Treated identically. Only one channel. Device name "KEYBOARD:". Filename and unit number ignored. No EXOS variables to set before open.

CLOSE/DESTROY CHANNEL - Treated identically.

READ CHARACTER/BLOCK - Returns ASCII key code or characters of function key string.

WRITE CHARACTER/BLOCK - Not supported

READ STATUS - C=0 if key has been pressed, C=1 if not.

SET STATUS - Not supported.

```
SPECIAL FUNCTION  -  @@FKEY = 8  Program function key
                     @@JOY  = 9  Direct joystick read
```

4.2 EXOS Variables

```
DELAY_KEY   -  Delay until auto repeat starts
RATE_KEY    -  Rate of autorepeating
CLICK_KEY   -  Zero to enable key click
LOCK_KEY    -  Current keyboard lock mode
STOP_IRQ    -  Zero to enable stop key software interrupts
KEY_IRQ     -  Zero to enable normal key interrupts
```

4.3 Software Interrupt Codes

```
  ?FKEY    = 10h  \
    .        .     \ Triggered by pressing function key
    .        .     / with null string programmed in.
?FKEY+15   = 1Fh  /

  ?STOP    = 20h  Triggered by pressing STOP key if
                      STOP_IRQ=0.  No key code returned.

  ?KEY     = 21h  Triggered by pressing any key if KEY_IRQ=0.
                      Key code returned as well.
```

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

1. Introduction

The serial interface and network are provided as two separate EXOS device drivers as far as the user is concerned, with device names "SERIAL:" and "NET:" respectively. However since they share the same hardware only one of the devices can be supported at a time. So if the user wishes to open a channel to the serial device he must first close any channels open to the network, and vice versa.

Only one serial channel may exist at a time, while any number of channels may be opened to the network device provided there is sufficient RAM for the 512 bytes of buffer for each channel. All channels opened to the network or the serial device support both input and output.

2. Hardware

The serial/network hardware consists of two outputs and two inputs. The outputs are the bottom two bits of an output port (port 0B7h) as below. The other bits of this output port are not used.

b0 - DATA OUT
b1 - STATUS OUT

Each of these outputs is connected to an open collector inverter with a pullup resistor. Thus for example setting bit 1 of this port will pull the STATUS OUT line low. Clearing this bit will allow the STATUS OUT line to float high unless any other connected machine is pulling it low.

The two inputs are available as bits of a general purpose input port. The other bits are used for cassette inputs and various other things. The port number is 0B6h and the relevant bits are:

b4 - DATA IN
b5 - STATUS IN.

For use as a serial interface these inputs and outputs are used separately to provide half duplex communication (data can be sent either way but only one way at a time). For use as a network the STATUS IN and STATUS OUT lines are joined together, as are DATA OUT and DATA IN.

NOTE: Throughout this document the signal levels referred to are the actual levels on the external lines. On the Enterprise, both inputs and outputs are inverted between these external lines and the appropriate bits of the ports, so the actual values of the bits will be clear for a high line and set for a low line.

3. Serial Device

3.1 Low-level Operation

The serial device uses five wires - DATA IN, DATA OUT, STATUS IN, STATUS OUT and REF (which may be used as an offset signal reference instead of the 0 Volt GROUND line). This allows independent handshaking on input and output. The device supports both read and write EXOS calls. When it is called with a "read character" function call it sets the STATUS OUT line high (it is normally held low). This signals the sending device that it can send a character. It then monitors the DATA IN line (which is also normally low) until it goes high signifying a start bit. The bits of the character can then be read in, possibly with a parity bit if that is selected (see later).

The serial driver handles a small buffer for incoming characters. After one character has been read, the DATA line is monitored for a short time to see if the sending machine has any more to transmit. If another character is sent, it will be read in and buffered. Up to sixteen characters may be read and stored if they are immediately available. Once the buffer is filled, or if no further start bit is detected within the timeout period, the STATUS OUT line is pulled low again preparatory to returning to the user program. However, some devices are rather slow in responding to handshaking lines, so the DATA IN line is checked for a short time afterwards to ensure that no more characters are being sent. Any spurious characters which are received can be buffered (there is an additional eight-byte overflow in case the main buffer is full). Stored characters are supplied to the user one at a time when "read character" is called, so this buffering is transparent.

"Write character" is simpler than read character since there is no problem of the other end of the connection misbehaving (ie sending extra characters). To send a character the serial driver monitors the STATUS IN line until it is high (which it may be already if the receiver is ready). Then the DATA OUT line is changed from its quiescent low level to high for the start bit. The bits of the data are then sent, followed by a parity bit (if parity is selected) and then the required number of stop bits (DATA OUT Low).

3.2 Use of the Serial Device

3.2.1 Read and Write Instructions

The serial device supports the normal EXOS read and write instructions for single characters or blocks. Data is not interpreted in any way, so machine code can be sent just as easily as ASCII text.

3.2.2 Baud Rate Selection

The EXOS variable BAUD_SER governs the baud rate, which applies both to input and output. Before opening the serial channel the user should set it to the appropriate value for the required rate, according to the following codes:

```
 0 => 50 baud          1 => 75 baud
 2 => 110 baud         3 => 134.5 baud
 4 => 150 baud         5 => 200 baud
 6 => 300 baud         7 => 600 baud
 8 => 1200 baud        9 => 1800 baud
10 => 2400 baud       11 => 3600 baud
12 => 4800 baud       13 => 7200 baud
14 => 9600 baud       15 => 9600 baud
```

The default setting is 15 (9600 baud). Numbers greater than 15 are reduced modulo 16 before interpretation.

3.2.3 Word Format Selection

The EXOS variable FORM_SER, defines the word format which is used for both input and output. Certain bits are interpreted as follows:

```
    b0 - Number of data bits:  Clear => 8 bits
                               Set   => 7 bits
    b1 - Parity enable.  Clear for no parity.
    b2 - Parity select (Ignored if b2 is clear).
                               Clear =>  even parity
                               Set   =>  odd parity
    b3 - Number of stop bits:  Clear => two stop bits
                               Set   => one stop bit
b4..b7 - Not used, must be zero.
```

The default setting is zero which selects 8 data bits, no parity and two stop bits.

Note that the data bits are sent least significant first, and if 7 data bits are selected then bits 0 to 6 of the byte will be sent and bit 7 will be ignored. On reception bit 7 will be cleared.

4. Network Device

4.1 Data Transaction Protocols

The network can be used in two modes: directed and broadcast. In directed mode, one machine sends data to a second machine and all other machines ignore it. In broadcast mode, one machine sends data to all the other machines at once. Each machine on the network is identified with a unique address in the range 1 to 32.

Each block that is sent consists of a header which may or may not be followed by a block of data bytes. The header includes a synchronisation pattern, the source and destination addresses and a type byte, of which the latter contains a flag determining whether any data bytes are to follow. In every header there is also a count of the number of data bytes in the block, although this is ignored if the type byte specifies that no data at all will be sent.

4.1.1 Broadcast Protocol

The header of a block which is being broadcast contains a destination address of zero. The block will be received by all machines which are listening and there is no method of determining whether it was read correctly or not.

This lack of handshaking introduces problems if some of the machines had interrupts disabled at the time of the broadcast, and thus arrive at the network interrupt handler once transmission has already started, or even after it has finished. In order to avoid possible confusion brought on by receiving only part of a block, the destination machines check for the synchronisation pattern which is supplied in the block header. The header consists of a repeating four-byte block, which continues for long enough to give the receiver time to be ready in most cases while obviously being kept reasonably short in order to save time.

4.1.2 Directed Data Protocol

As Directed Data is destined for just one machine, the sending machine can wait for acknowledgement in order to ensure that the destination is listening, and to confirm that the data block is received without error.

When a computer reads a block header which has its own number as the destination address, and is prepared to accept the block (see later), then it sends back an acknowledgement to the source machine. This is simply a signal on the DATA line to show that it is ready to receive; the absence of the signal within a given time (about 4 bit periods at the selected baud rate) is interpreted by the source machine as an error.

If any data bytes are to be sent, these are transmitted once the header has been acknowledged. After the complete data block has been received, the destination machine must carry out the checksum calculation and either confirm the data by sending another acknowledgement signal, or reject it by not responding.

Error Response

When a destination machine finds an error it returns immediately from the network interrupt routine without setting any interrupt flags (see later). The source machine, when it finds no acknowledgement signal after sending a header or a data block, retries as follows:

1. Release network *

2. Wait for long random delay, of the order of a quarter of a second.

3. Try to gain control of network and send again

* Note: It is extremely important that under no circumstances should an error occur which causes a machine to hang irretrievably while it is in control of the network, as this would also hang the network itself and thus any other machines which are trying to use it. Any time a machine is waiting indefinitely for a signal on the network lines, pressing the STOP key will regain user control.

4.1.3 Accepting Data Blocks

A machine is only prepared to receive transactions from another computer on the network if there is a suitable channel open to the network driver:

When a channel is opened by the user a remote address number is given as the unit number. If this is zero then blocks will be accepted from anywhere. If it is non-zero then only blocks from that specific network address will be accepted on this channel, any number of such non-zero address channels may be opened provided they all have different addresses.

If a non-specific channel is opened it will only receive data from machines which are not explicitly served by an individual channel. Only one non-specific channel may be open at a time.

4.2 Low-level Network Operation

4.2.1 Hardware Connections

The network driver is rather more complicated than the serial interface driver because it has to include protocols to avoid collisions. It uses the same hardware as the serial driver. All machines on the network are joined by three wires: GROUND, DATA and STATUS. On each machine the DATA line is connected to both DATA OUT and DATA IN, and the STATUS line is connected to both STATUS OUT and STATUS IN. This allows the machine to pull either line low (the outputs are open collector) and also to monitor the level on each line.

The STATUS IN line is also connected to the external interrupt input so the status line going low can trigger an interrupt. This is how the machine can respond to data sent down the network asynchronously.

4.2.2 Obtaining Control of the Network

When a machine wishes to send data to another machine, or to broadcast it, it must first get control of the network. Only one machine can be in control of the network at a time and the protocol used for obtaining control is designed to ensure that collisions (two machines taking control at the same time) do not occur. The penalty for this is a slight loss of speed and a priority ordering of machines on the network.

There is a timing constant C defined, which corresponds to a delay of the order of one millisecond. When a machine wants control of the network it must follow this procedure:

1. Both STATUS OUT and DATA OUT should be left high whenever this machine does not have control of the network.

2. If the STATUS line is high go to step 4

3. Wait until the STATUS line is high and remains High for a period of RND*C where RND is a random number in the range 1 to 16. If the STATUS line does not remain high for the required time then repeat this step with a new random number.

4. Pull the STATUS line low.

5. Wait for a period of C*ADDR where ADDR is the address of this machine on the network, constantly monitoring the DATA line. If DATA goes low for any time during this interval then release the STATUS line (set it high again) and return to step 3.

5. Pull the DATA line low and then proceed to send the block header (see later).

Throughout a network transmission, interrupts are disabled because of the timing considerations.

4.2.3 Attracting Attention - Protocol

Once a machine has secured itself control of the network, it can start the transmission of the data block.

First of all, it has to attract the attention of its intended audience. This is done by repeatedly sending a header consisting of a synchronisation byte, the destination and source addresses of this block and a type byte. The bytes are sent in the order sync/dest/source/type/sync.... and are terminated with the ones complement of the dest byte in the position of the next sync byte. The format of these bytes is as follows:

Machine address bytes

The destination address is the number of the machine on the network to which this block is being sent; if it is zero then it indicates that it is a broadcast block, which all machines should receive. The source address is the number of the machine which is sending the block.

Byte format:

b0..b5 - machine address: 1 to 32, or 0 for broadcast (0 only valid as a destination address)

b6 - source/destination selection: 0 => addr=source
1 => addr=dest

b7 - complement of bit 6

Synchronisation byte 00000000B

The synchronisation byte is required because fast interrupt response is not guaranteed; the STATUS line can be brought Low at any time, perhaps while interrupts are disabled on the required destination machine. The synchronisation method is detailed below.

Type byte

The type byte contains information defining what kind of transaction this is, including whether or not any data bytes will be sent after the header. Its format is as follows:

b0 - data flag: 0 => no data to follow header
1 => data block to follow

b1..b4 - not defined - must be zero

b5 - end-of-record flag:
1 => end of record
If the block was sent as a result of a Flush or Close command, this bit is set to force the network Read Character routine to return the 'End of file' status, thus identifying the end of a given message or file. The next Read Character or Read Status call will revert to the 'Character not available' state, unless the end-of-file flag is also set...

b6 - end-of-file flag:
1 => end of file
Only ever set in conjunction with the end-of-record flag, it implies that the sending machine has closed its channel to this computer. All subsequent requests for channel status or for further characters will return the 'End of file' condition, until another block of data is received on this channel.

b7 - Type byte flag - must ALWAYS be set to 1, so that the synchronisation byte is guaranteed to be the only string of eight zero bits in the header.

4.2.4 Destination Machine Synchronisation

As stated above, there is some problem in synchronising the destination machine, which stems from the uncertainty of catching the machine while its external interrupt is enabled. This is now explained in detail:

If external interrupts are disabled, a latch will record the interrupt transition. The interrupt will then take place as soon as the input is re-enabled, and the destination machine will immediately start looking at the DATA line.

The source machine is not expected to wait for long enough to ensure that all machines will be listening, but instead begins to output its attention-grabbing sequence as soon as it is ready - in the hope that all relevant machines will catch on before that sequence finishes. The protocol must therefore cope with a destination machine starting to read the DATA line at any time during this period.

RS232 character framing is by means of DATA line levels, so any transition from 1 to 0 can be interpreted as a start bit. This means that a computer which starts to listen to the DATA line at an arbitrary moment may pick up what it thinks is a start bit when in fact it has simply found a changing level in the middle of a character. The purpose of the synchronisation byte in the header sequence is to make sure that the character framing is correctly aligned as quickly as possible.

The method chosen for achieving this is to send a synchronisation character made up of all zero bits, and to tell the destination machine to check every start bit transition until it finds one which is followed by eight zero bits and a stop bit. The next 'start bit' is then guaranteed to be the true start of a character.

The above specifications require that any aspiring destination machine should follow this algorithm when it enters the interrupt routine:

1. If STATUS line is high, return. (This would occur if the response had been so slow that the transfer was all over, or had been abandoned due to lack of interest)

2. Wait until the DATA line is low

3. Wait for DATA line to go high (possible start bit)

4. Delay for required time to synchronise to middle of bits

5. Continue to read DATA at one-bit intervals until a low bit is found

6. If any number other than nine high bits were found (including the start bit) go to 4

7. Store the synchronisation byte, and the next three bytes which are received, in scratch memory. These four values are the header pattern, and should be sent repeatedly by the source machine.

8. Check that the destination byte is valid, and that it holds either this machine number or the value zero. Return if this is not the case.

9. Check that the source byte is valid, and that there is a channel open which can serve the specified machine - ie. either a specific channel for that address, or a non-specific channel. Otherwise return.

10. Check that the type byte is valid.

11. Continue to read bytes. Check that each group of four matches the values stored in step 7, and continue to loop until a value does not tally.

12. The value which disagrees should be the sync byte, and it should contain instead the ones complement of the dest byte. If this is not the case, return.

13. Read the next two bytes. If they are 1's complement of each other then the latter is the byte count for the following data block. Else return.

14. If the destination address was non-zero (ie. transaction is directed rather than broadcast), pull DATA low for about 1ms as acknowledgement, then release it.

NOTE: Whenever a byte is read, STATUS is checked for still being held low by the source machine. If it goes high at any time, the link is immediately assumed broken.

4.2.5 Sending the Data Bytes

When the destination machine pulls the DATA line low, and leaves it there for at least half a millisecond, that is the signal to the source machine to start sending any data which may be included in this transaction. When DATA is released, the sender pulls it low again immediately then gets on with sending the data block.

The format of the data block is:

```
 ---------------- --------- ----------
|   Data Bytes   | CRC low | CRC high |
 ---------------- --------- ----------
```

The byte count which was received in the header sequence is a true count of the number of data bytes in the block. A value of zero means 256 bytes.

The two CRC bytes are a cyclic redundancy check calculated on all data bytes in the block (but not the header). The CRC is evaluated as a 16-bit value, which is re-calculated when the data block is received. If the CRC calculated on reception is not the same as the value which was sent then the block is assumed to be garbage.

The algorithm for calculating the CRC is the same as that used for the cassette driver but is explained here as well: For purposes of the CRC calculation the data block is regarded as a bit stream. Each time a data bit (not a start or stop bit) is sent, a 16 bit CRC register is updated. This is done in the serialisation routine which is shared with the serial interface driver, so the CRC is also calculated on data sent or received through the serial handler even though this is not required.

The CRC register is a 16 bit value which is initialised to zero at the start of the data block, and then when each bit is sent or received the following operations are performed:

1. XOR the new bit with the most significant bit of the CRC register and set the carry to this value.

2. If the carry is set then XOR the register with 0810h.

3. Rotate the register one bit left, moving the carry into the least significant bit.

4.2.6 Data Transmission Format

All network blocks are sent as a series of bytes in the same format as the serial interface driver sends them, using a word format of eight data bits, two stop bits and no parity. The characters are transmitted at the current1y selected baud rate, which defaults to 9600 baud.

4.3 Using the Network

4.3.1 Address Determination

The above protocols require that each machine on the network has a unique address, so the network handler refuses to open channels onto the network unless the computer has been given its address.

The address is held in the EXOS variable ADDR_NET, and must be set by the user before attempting any network operations. At power-on or cold reset it is set to zero, which is invalid as a network machine number.

4.3.2 Opening Channels to the Network

As mentioned before, each channel which is opened to the network must specify a remote machine number for which the channel is reserved, although an address of zero will allow data blocks to be received from any machine. This number is given in the call to the EXOS channel opening function as the unit number (see EXOS kernel specification). Any filename given is ignored.

Note the clear distinction between the channel number and the network address of the machine which that channel serves. The two are completely independent; it is up to the user to keep track of which channel serves which machine when he wishes to output to the network. For input, both values are made available to him by the interrupt handling code (see below).

4.3.3 Interrupts

When the first channel is opened to the network, the external interrupts are enabled and will remain so until the last channel is closed. While external interrupts are enabled, any transition from high to low on the STATUS line will cause the network interrupt service routine to be called. A particular consequence of this is that opening a serial channel (for which the quiescent line levels are low) on a machine which is connected to a network, will cause any machines using the network to be interrupted, and to hang in anticipation of a data block.

4.3.4 Buffers

Each channel opened to the network sets up two buffers of 256 bytes length; a receive buffer and a transmit buffer.

The receive buffer is filled with data bytes which arrive in blocks on the network, and can hold just one block at a time (no matter how short that block is; even a block of just one byte effectively 'fills' the receive buffer until it has been read or cleared). The user reads from the buffer until it is empty, when it will be able to accept further blocks from the network. The buffers of each channel are independent, so blocks can be received from any machine provided that the receive buffer which serves it is free.

The transmit buffer is filled by the user, and sent as complete blocks onto the network. It can be forced onto the network at any time by using a "flush" special function call, or will be sent automatically when the 256th byte is written to it. The Transmit buffer will also be flushed automatically if the channel is closed, thus enabling files of any length (from 1 byte upwards) to be sent to another machine while the buffering remains transparent.

4.3.5 Read and Write Functions

The network device supports the usual EXOS read and write character function calls and the block read and write calls.

The reception of blocks from remote machines is done by an interrupt service routine, which is invoked when the STATUS line goes low. The machine will remain in its interrupt routine watching for the DATA line to go low and then read in the header. If the header is read succesfully, and the block is one which this machine wants to receive, then the data is read into the receive buffer for the channel serving the given remote machine. Each channel can buffer one block at a time in its receive buffer.

The network driver can cause a software interrupt when a block is successfully received from the network. An EXOS variable called NET_IRQ is provided to switch this function on and off. If NET_IRQ is zero then software interrupts are enabled, and the occurence of a successful network interrupt will cause the value ?NET to be placed in the variable FLAG_SOFT_IRQ.

The EXOS variable CHAN_NET is used to pass to the user, the channel number from which buffered data can be read. CHAN_NET is only updated when a character is read from the channel which it specifies, or if that channel is closed. It is then changed to the channel number of the lowest numbered machine which has caused an interrupt. So if machines 2 and 10 send a block each while data sent by machine 5 is still waiting to be read, then when CHAN_NET is changed it will point to the channel for machine 2 rather than 10 whichever interrupted first. Machine 10 will be serviced later, as soon as it becomes the lowest number awaiting attention. This allows specific machines to be given priority - a teacher operating from machine 1 will almost always have his message received in preference to a message from another machine, while broadcast messages are given the highest priority of all. If no data is available CHAN_NET holds the value 255.

Whenever CHAN_NET is updated, the EXOS variable MACH_NET is also altered to hold the number of the machine which caused the interrupt. This is vital when a block is received on the non-specific channel, since there would otherwise be no way of telling which machine sent it. In other cases it is simply useful - as stated above, the user would normally be expected to keep his own records of which channel serves which machine.

An application program requests bytes by calling the EXOS RDCH or RDBLK routines, and is given bytes from the receive buffer if they are available. If not, there are two possible responses: If an End-of-record flag was received in a header for this channel, an 'End of file' condition is returned. Otherwise, the Read routine will halt until this channel receives data from the network.

The "read channel status" function call is supported, so the user can check for the 'End of file' or 'Character not available' conditions before trying to read a character.

Writing to the network is carried out as a series of WRCH or WRBLK function calls. If an error is encountered when the buffer is to be sent, the network handler will continuously retry at intervals of between quarter and half a second. This can be stopped by pressing the STOP key, which will also cause the buffer contents to be cleared before the handler returns. Note that this error condition can only be detected for directed data blocks; broadcast data does not require acknowledgement and therefore cannot check that the data was received properly.

5. Quick Reference Summary

5.1 SERIAL - EXOS calls

OPEN/CREATE CHANNEL	- Treated identically. Only one channel, and no network channel. Device name "SERIAL:". Filename and unit number ignored. No EXOS variables to be set before open.
CLOSE/DESTROY CHANNEL	- Treated identically.
READ CHARACTER/BLOCK	- Reads characters from serial input, some buffering.
WRITE CHARACTER/BLOCK	- Writes bytes without interpretation.
READ STATUS	- Always character ready (C=0).
SET STATUS	- Not supported.
SPECIAL FUNCTION	- No special functions.

5.2 NETWORK - EXOS calls

OPEN/CREATE CHANNEL - Treated identically. Multiple channels allowed, but no serial channel. Unit number defines destination machine number, filename ignored. Device name "NET:". EXOS variable ADDR_NET must be set before open.

CLOSE/DESTROY CHANNEL - Treated identically. Will try to send any buffered data.

READ CHARACTER/BLOCK - Reads characters from buffer if avaialable, else returns EOF or waits.

WRITE CHARACTER/BLOCK - Puts bytes in buffer. Written to network when FLUSH special function call is done, or channel is closed, or buffer is full.

READ STATUS	-	C=0	if character in buffer.
		C=0FFh	if end of record.
		C=1	if buffer empty.
SET STATUS	-	Not supported.	
SPECIAL FUNCTION	-	@@FLSH = 16	Send buffered data with end-of-record flag set.
		@@CLR = 17	Clear send and receive buffers.

5.3 EXOS Variables

FORM_SET	- Serial format. Set to zero by network.
BAUD_SER	- Serial and network baud rate.
ADDR_NET	- Network address of this machine.
CHAN_NET	- Channel on which network block received.
MACH_NET	- Machine from which network block received.

NET_IRQ - Zero to enable network software interrupts

5.4 Software Interrupt Codes

?NET = 30h Occurs when a data block is received by a network channel if NET_IRQ is zero.

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

1. General Device Interface

The printer driver is a very simple device which just sends characters to a printer (or other device) using the built in centronics type parallel interface.

Only one channel at a time may be open to the printer driver, if an attempt is made to open a second channel then an error (.2NDCH) will be returned. A channel can be opened by giving the device name "PRINTER:", any filename or unit number is ignored.

Having opened a channel characters can be written using either the single characer write or the block write function call. The characters will be sent without any interpretation at all, and all 8 bits are sent.

2. Hardware Details

The hardware consists of one eight bit output port for the parallel data (port 0B6h), one other output bit for a data strobe (bit 4 of port 0B5h) and one input bit as a ready signal (bit 3 of port 0B6h).

To send a byte the printer driver outputs the character to the data port and then waits until the ready signal goes low. When the ready signal is low it strobes the data by setting the data strobe low for a few microseconds and then setting it high again (it is normally high when not in use). This completes the sending of a character.

The other bits of output port 0B5h are used for various control operations such as scanning the keyboard, and controlling remote control relays. A variable (PORTB5) which is at a fixed address defines the current state of this port and the printer driver ensures that all other bits of the port are maintained in their correct state.

3. Quick Reference Summary - EXOS calls

OPEN/CREATE CHANNEL - Treated identically. Only one channel. Device name "PRINTER:". Filename and unit number ignored. No EXOS variables to be set before open.

CLOSE/DESTROY CHANNEL	-	Treated identically.
READ CHARACTER/BLOCK	-	Not supported.
WRITE CHARACTER/BLOCK	-	Writes bytes without interpretation.
READ STATUS	-	Not supported.
SET STATUS	-	Not supported.
SPECIAL FUNCTION	-	No special functions.

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

The video and keyboard channels specified in VID_EDIT and KEY_EDIT must be opened before opening the editor channel. The video page must be a text mode and must be at least 3 rows by 4 characters. The editor determines the size and mode of the video page when the channel is opened and returns an error (.EVID) if it is unsuitable. Note that the editor does not display the video page on the screen, it is up to the applications program to take care of this.

The actual size of the editor's buffer which is available for storing text is:

256*BUF_EDIT + n

where 'n' is between zero and 255 and depends on the width of the video page (space reserved for the ruler line) and the exact size of the editor's variable area. The valid range for BUF_EDIT is thus zero to 254.

The editor will work with any size buffer but it is sensible to ensure that it is at least as big as the video page so that the editor is always capable of displaying a full page. The editor stores lines as variable length in its buffer so, since short lines are common, it generally manages to store more than the calculated minimum number of lines.

3. General Editor Features

3.1 The Editor's Text Buffer

As mentioned before the editor has a text buffer in which it stores its text, and the video page just provides a window onto part of this buffer. Text is stored in the buffer on a line orientated basis. Each line has a three byte line header containing certain flags and margin information. This is followed by the text of the line itself stored in ASCII. The line is terminated by a special character which indicates whether it is the last line in a paragraph and whether it is the last line in the buffer.

The lines are stored with variable length so if a line only has four characters on it, followed by 36 spaces then the 36 spaces are not stored. This improves buffer usage very significantly since in general short lines are quite common. There is no limit to the number of lines in the buffer other than the total size of the buffer.

1. Introduction

All the other built in device drivers provide an interface to some aspect of the hardware such as the cassette I/O circuitry or the DAVE chip. The editor however does not interface directly to any hardware, instead it provides a higher level user interface to two of the other built in drivers - the video driver and the keyboard driver.

An editor channel can be thought of as an intelligent, full screen editing terminal handler. It can be used by an applications program to provide all of its general purpose communication with a user. For example the IS-BASIC cartridge does all of its screen and keyboard I/O through an editor channel. BASIC will be used frequently in this document as an example of how to use the editor.

The editor can support any number of channels open to it at a time, each channel corresponds to a separate "document" which is being edited. The word document here is used loosely since for example the editor channel used by BASIC is referred to as a document although it is actually a collection of BASIC commands, program listings, error messages, program output, etc.

Each editor channel has video channel and a keyboard channel associated with it. Different editor channels can share the same keyboard channel (which is essential since the keyboard driver only allows one channel to be open to it), but must have separate video channels.

Each editor channel also has an area of channel RAM which it uses for a text buffer. This buffer can be any size from a few hundred bytes to just under 16k and will typically be a few kilobytes. Text can be entered into the editor's buffer either from the applications program or from the keyboard. The editor writes characters to the video page in such a way that it is kept updated to form a "window" onto the text buffer. This is not a true window since the video page has its own copy of the text it is displaying.

2. Opening Channels

An editor channel can be opened by giving the device name "EDITOR:", any filename or unit number is ignored. Before opening an editor channel, three EXOS variables must be set up. These are:

```
VID_EDIT  -  Channel  number  of video page.
KEY_EDIT  -  Channel  number of keyboard channel.
BUF_EDIT  -  Size of editor buffer in units of 256 bytes.
```


The video and keyboard channels specified in VID_EDIT and KEY_EDIT must be opened before opening the editor channel. The video page must be a text mode and must be at least 3 rows by 4 characters. The editor determines the size and mode of the video page when the channel is opened and returns an error (.EVID) if it is unsuitable. Note that the editor does not display the video page on the screen, it is up to the applications program to take care of this.

The actual size of the editor's buffer which is available for storing text is:

256*BUF_EDIT + n

where 'n' is between zero and 255 and depends on the width of the video page (space reserved for the ruler line) and the exact size of the editor's variable area. The valid range for BUF_EDIT is thus zero to 254.

The editor will work with any size buffer but it is sensible to ensure that it is at least as big as the video page so that the editor is always capable of displaying a full page. The editor stores lines as variable length in its buffer so, since short lines are common, it generally manages to store more than the calculated minimum number of lines.

3. General Editor Features

3.1 The Editor's Text Buffer

As mentioned before the editor has a text buffer in which it stores its text, and the video page just provides a window onto part of this buffer. Text is stored in the buffer on a line orientated basis. Each line has a three byte line header containing certain flags and margin information. This is followed by the text of the line itself stored in ASCII. The line is terminated by a special character which indicates whether it is the last line in a paragraph and whether it is the last line in the buffer.

The lines are stored with variable length so if a line only has four characters on it, followed by 36 spaces then the 36 spaces are not stored. This improves buffer usage very significantly since in general short lines are quite common. There is no limit to the number of lines in the buffer other than the total size of the buffer.

The buffer is arranged as a circular buffer so the start of the text may be anywhere in the buffer, and the end of the buffer may occur at any point in the text with the text continuing again at the start of the buffer. This avoids ever having to move the whole text up or down in the buffer.

3.2 When the Buffer Becomes Full

When new text is entered into the buffer, either at the end or into the middle of the buffer this clearly uses up buffer space. Eventually the buffer may become full. In fact it is generally the case when using BASIC that the buffer is nearly full most of the time, as it contains previous commands and so on which have scrolled off the screen.

Whenever there are less than 100 bytes spare in the buffer the editor displays a number on the right hand side of the status line indicating the number of free bytes. As characters are typed in, this number will get smaller until it eventually reaches zero. This number is only displayed when waiting for keyboard input from the user, so for example the number cannot be seen when BASIC is listing a program even though the buffer may be full.

When the buffer is full and another character is typed in (or written by the applications program), the editor has to delete some of the existing text to make room. It always deletes a whole line of text and it generally deletes the first line since this will normally be the oldest and least useful one. If the first line of the text is displayed on the video page then it deletes the last line instead, to avoid deleting text which is displayed.

If the editor buffer is very small or there are some very long lines, then every line may be displayed at once, so the editor has no choice but to delete a line which is displayed. In this case the editor deletes the last line unless the cursor is on the last line, in which case it deletes the first line and scrolls the page up.

Note that because the editor buffer is circular, deleting this line of text does not involve moving the whole text up to fill the gap (at least not usually).

3.3 Margins and the Ruler Line

Each line in the buffer has its own individual left margin position. When a new line is created it will be given a left margin equal to the current left margin setting which can be displayed on a ruler line at the top of the video page. There is also a right margin which is displayed on the ruler line. In general text can only be entered in between these margin settings although there is the facility of temporarily releasing the margins.

3.4 Paragraphs

Lines in the editor's buffer are grouped together in paragraphs. When the user presses ENTER (or a CR is received from the applications program) this marks the current line as the end of a paragraph. It also moves the cursor to the start of the next line which will be the start of a new paragraph (and may have the side effect of sending text to the applications program - see later).

If the user types a very long line then the editor will split the line at a sensible point (using a process called word wrap described in the next section) to give two lines. The first line will be terminated by a soft carriage return marker to indicate that it is not the end of a paragraph. In this way long paragraphs can be built up.

There is no indication on the screen of where paragraphs start and end but some of the editing functions operate on paragraphs, and the paragraph is the basic unit for sending text back to the applications program.

3.4 Word Wrap

Word wrap is the process which decides where to split a line which is too long. When a character is typed outside the margins (assuming that margins are not released) then the editor searches back to find the start of the word which contained that character and moves the whole of that word onto the start of a new line.

This process is done with all text received from the applications program as well as that typed at the keyboard from the user. Thus BASIC listings are subject to word wrap so keywords and variable names etc. will not be split in half.

3.5 Long Lines

Although a very long line which is typed in will be split by word wrap, it is possible to create a long line by inserting characters into the middle of a line, which will push the rest of the line to the right. In this way it is possible to create a line which is too long to be all displayed on the video page. This fact is marked by a red angle bracket (">") on the extreme right hand end of the line on the video page. This is an overflow marker.

The part of the line which has gone off the page cannot be accessed although it is remembered by the editor. The line can only be accessed by reformatting it to bring it back onto the page. There are various editing commands which can do this described later on.

3.6 Flashing Cursor

The EXOS video driver which the editor uses only provides a static non-flashing cursor display, although this can be turned on and off. The editor implements a flashing cursor by simply turning the video page's cursor on and off regularly while it is waiting for input from the user. It always ensures that the cursor is switched off when it is doing any editing functions, since these can result in the cursor having to move all over the screen and it is rather messy if the cursor can be seen doing this.

The editor also switches the cursor off whenever it returns to the applications program and it remains off when the applications program is writing characters to the editor. This results in a nice clean cursor display, where the flashing cursor always means that the editor is waiting for the user to type some input.

4. Writing to the Editor

Any characters in the range 20h to 9Fh, written to the editor are regarded as printing characters and are put into the text buffer at the current cursor position and displayed on the video page. They are subject to word wrap as described above.

The editor will also interpret the following control codes. All codes in the range 00h to 1Fh not mentioned here are ignored.

00h (NUL) - Writes a null to video to check it is still OK.
09h (TAB) - Move to, or insert spaces to, next tab stop.
0Ah (LF) - Ignored.
0Dh (CR) - Goes to start of new line (equivalent to CR-LF).
18h (^X) - Set left margin at cursor column.
19h (^Y) - Clear to end of line.
1Ah (^Z) - Clear whole buffer and screen and home cursor.
1Bh (ESC) - Starts escape sequence (see below)

The only escape sequence interpreted by the editor is to position the cursor at arbitrary co-ordinates. This is identical to the video driver escape sequence for this function and details can be found in the video driver specification. It positions the cursor at the specified co-ordinates of the video page, regardless of which portion of the editor's buffer is currently being displayed.

Codes in the range 0A0h to 0FFh are interpreted exactly as if they had been received from the keyboard. These provide various editing functions and cursor movement. They are described in detail in the section on editing functions.

5. Reading From the Editor

When the editor receives a read character function call, it examines the EXOS variable FLG_EDIT. This byte contains a series of flags which control the response of the editor to this read character call. The editor's action will first be described in general terms without reference to the individual flags. The effect of each flag will then be described in detail.

5.1 Basic Editor Read Action

Assuming for now a typical setting of the flags in FLG_EDIT, when the editor receives a read character call it interprets this as a request to send a line of text back to the applications program. It does not return a character immediately but records the state of the flags and enters its main editing loop which provides the actual editing facility to the user. It is only while in this loop that the cursor on the video page will flash. A flashing cursor thus indicates that the editor is waiting for a key to be pressed.

This loop reads a character from the keyboard, responds to it and then loops back for another one. This allows the user to type in text which will be inserted into the editor's text buffer and displayed on the video page, and to carry out various editing functions. The details of the editing functions are given later. There are two keys which are of importance here - ESCAPE and ENTER.

If ESCAPE (ASCII code 01Bh) is received from the keyboard then this character will immediately be returned to the applications program as the response to the read character call, regardless of the state of any of the FLG_EDIT flags. This provides a way of interrupting a line input operation.

If ENTER (ASCII code 0Dh) is pressed then this is a command to the editor to begin sending text back to the applications program. The details of how much text is sent are determined by the flags and will be described below. An internal editor flag is set to indicate that it is in the process of sending text, and the first character is returned to the applications program. When the applications program makes another read character call the internal flag is still set, so instead of entering the editing loop, it simply returns the next character of the requested text immediately. This continues until all the required text has been sent at which time the internal flag is cleared so the next read character call will again enter the editing loop.

It is up to the applications program to recognize when it has read all the characters to avoid re-starting a new read operation. How to recognize this depends on the setting of the editor flags and is described later.

Note that the editor flags are sampled once when the first read character call is made and then not again until all the text has been sent. Changing them in the meantime will therefore have no effect on the read which is in progress. If a write character call is mode while a read is in progress then the internal flag is cleared so that read will be aborted. This also applies if an special function calls are made.

5.2 The Editor Read Flags in Detail

The assignment of bits in the FLG_EDIT EXOS variable is:

MSB	bit-7	-	SEND NOW
	bit-6	-	SEND ALL
	bit-5	-	NO READ
	bit-4	-	NO SOFT
	bit-3	-	NO PROMPT
	bit-2	-	AUTO ERA
	bit-1	-	not used
LSB	bit-0	-	not used

5.2.1 The SEND NOW Flag (bit-7)

If this flag is set then the editor will start returning text immediately, without reading from the keyboard at all. If it is clear then the main editing loop will be entered and no text will be returned until the user types ENTER. In either case the amount of text returned is the same and depends on the setting of the other flags.

5.2.2 The SEND ALL Flag (bit-6)

This is the main flag which determines how much text is sent. If it is clear then the paragraph containing the cursor will be sent. If it is set then the whole editor buffer will be sent.

In the first case the applications program will be sent the characters of the paragraph one by one terminated with a CR and then an LF. This CR-LF can be used by the applications program to determine when to stop reading (beware if NO SOFT is clear! - see below). The cursor will be left on the first character of the next paragraph with a new (empty) paragraph being created if that was the last one.

If SEND ALL is set then the entire buffer will be sent. This will include CR-LF sequences at least between each paragraph (again see NO SOFT flag below) so this cannot be used to indicate the end of the text. Instead, after the last CR-LF has been sent (the last characters sent will always be CR-LF), the next character read will produce a .EOF (end of file) error. This error will only be received for one character so the applications program must notice it and stop reading. If another character was to be read then this would start the whole reading process again from the start of the buffer.

5.2.3 The NO READ Flag (bit-5)

If this flag is set then ENTER will not in fact return any characters at all to the applications program. The only way to get back to the applications program is thus to press ESCAPE. Although no characters are returned, the routine which selects which characters to send depending on the flags, is still executed. Thus the cursor is moved in the same way as if the text was returned. If SEND ALL is clear then pressing ENTER will just move to the start of the next paragraph, putting in a new line if it was at the end of the buffer. This will make the editor behave rather like a typewriter.

5.2.4 The NO SOFT Flag (bit-4)

This flag controls the sending of soft carriage returns and soft spaces. Soft carriage returns are those which separate successive lines of a paragraph. Soft spaces are those spaces which are inserted for justification, and also the spaces before the left margin of a line.

If this flag is clear then soft spaces are returned as normal ASCII spaces (20h) and soft carriage returns are returned as normal CR-LF sequences. If the flag is set then both of these are suppressed and no characters are returned for them.

Beware that if NO SOFT and SEND ALL are both clear then there is no way for the applications program to determine whether a CR-LF which it receives is the end of the paragraph, in which case it should stop reading, or simply the seperator between two lines of the paragraph, in which case it should continue. Therefore this combination of flags should be avoided - at least one of them should always be set.

5.2.5 The NO PROMPT Flag (bit-3)

This flag is normally clear. If it is set then the cursor position when the read operation was started is remembered. When it comes to returning text, if the cursor has not been moved out of the original paragraph, or to before the remembered position in the current paragraph, and if SEND ALL is clear, the the current paragraph will be sent back but starting from the character at the remembered cursor position rather than the start.

If the cursor has been moved out of these bounds then the whole paragraph will be returned as usual, or the whole editor buffer if SEND ALL is set.

This feature is used by BASIC when doing an INPUT command. It prints the prompt and the does an editor read with this flag set. When the paragraph is sent to BASIC the prompt will not be returned, just the response to it.

5.2.6 The AUTO ERA Flag (bit-2)

This flag is also normally clear. If it is set then, if the very first character typed at the keyboard is a printing character (as opposed to an editing function or cursor movement) then the current line will be cleared before responding to the key. This is provided mainly for BASIC to allow commands such as RUN to be typed on top of an existing line after editing a program. It may just conceivably be of some use to other applications programs.

5.3 Typical Flag Combinations.

The use of these flags can be rather confusing so this section discuses some examples of their use from IS-BASIC and the built in word processor (WP). This covers most useful combinations and certainly shows the use of each of the flags.

5.3.1 BASIC reading a command line. Flags = 000101xx

When BASIC is reading a command line from the editor it is expecting either an immediate mode command (such as RUN) or a new line starting with a line number to by typed. In either case it wants to read a single paragraph entered by the user. This might be newly typed by him or it might be a line which already exists in the editor buffer which he simply moves the cursor to and re-enters.

Clearly BASIC wants to wait for the user to type ENTER so the SEND NOW flag is clear. Only one paragraph, rather than the whole buffer is wanted so SEND ALL is clear and BASIC wants to actually be sent the text so NO READ is clear. BASIC is not interested in the breaks between lines in the paragraph (since one command line can over flow onto several screen lines) so NO SOFT is set. NO PROMPT is clear and AUTO ERA is set (as explained above).

5.3.2 BASIC doing an input command. Flags = 000110xx

When BASIC is doing an input command it also wants to read in a single paragraph so the SEND NOW, SEND ALL, NO READ and NO SOFT flags are all set the same as for reading a command. In this case however BASIC will write out a prompt and wants to read in just the response to that prompt, rather than having the prompt at the start of the paragraph which it receives. To do this it sets the NO PROMPT flag. The AUTO ERA flag is clear.

5.3.3 WP normal editing mode. Flags = 001xx0xx

In normal editing mode the word processor wants to let the user get on with his editing without data being sent to the word processor. The SEND NOW flag is clear to allow the user to do editing. The SEND ALL flag is clear and the NO READ flag is set to ensure that no characters are returned to the word processor when ENTER is pressed but the cursor will be moved to the start of the next paragraph. The NO SOFT and NO PROMPT flags are irrelevent when NO READ is set, and the AUTO ERA flag is clear.

This will have the effect of allowing the user to move all over his document, pressing ENTER and using any of the editing features. Only when ESCAPE is pressed will the word processor applications program be alerted. In fact the eight function keys have the ESC code as the first code in their programmed string so the word processor gets alerted when they are pressed as well.

5.3.4 WP Printing a Document. Flags = 11000xxx

When the word processor is asked to print a document it must read the whole of the editor's text buffer in order to print it. Also it wants to get the data immediately rather than waiting for the user to press ENTER. To achieve this SEND NOW and SEND ALL are both set. NO READ is of course clear or no text would be sent and NO SOFT is clear so that all spaces and soft carriage returns will be sent to ensure that the formatting of the printed document is correct. NO PROMPT is clear. AUTO ERA does not matter because SEND NOW is set so the user doesn't get a chance to press a key.

6. Editing Functions

The editor provides many editing and word processing features. These are carried out in response to the user typing an appropriate key on the keyboard, or by the same code being written from the applications program. The following sections describe each of the editing functions in some detail.

Some of the editing functions such as paragraph movement and reforming paragraphs, require fairly complex internal operations to be carried out. This often results in rather strange behaviour of the screen display, involving scrolling operations which might not be expected.

The four joystick directions and the INS, DEL and ERASE keys each have three different functions which are explained below. These functions are obtained by using the key alone, or with either the SHIFT or CTRL keys. In fact the CTRL function can also be obtained by using the ALT key.

6.1 Cursor Movement (The Joystick)

Cursor movement is the most fundamental operation for a full screen editor. On the Enterprise, cursor movement is carried out with the joystick in conjunction with the SHIFT and CTRL keys. The autorepeat on the joystick allows continuous cursor movement by just holding the joystick in one of its eight possible positions.

The cursor can be moved anywhere on the video page but cannot be moved off the page. If an attempt is made to move it off the top or bottom of the page then the display will scroll to bring more text from the buffer onto the page. This scrolling will stop when the start or end of the text is reached.

The cursor can be moved beyond the end of lines or outside the margin settings without the text being affected. However when a character is typed, extra spaces will be put in to fill up to the cursor position and word wrap may occur if the cursor is outside the margins.

Although only the four orthogonal directions of cursor movement are provided by the editor, diagonal movement is still possible. This is because if the joystick is moved to one of the diagonal positions, the keyboard driver autorepeat will return the appropriate two joystick codes alternately so the editor will execute them alternately. This is only useful for simple cursor movements, not for the shifted and controlled movements.

The possible cursor movements and their codes are:

Joystick Movement	Key Code	Function
UP	0B0h	Cursor up line
Shift-UP	0B1h	Cursor up page
Ctrl-UP	0B2h	Cursor up paragraph
DOWN	0B4h	Cursor down line
Shift-DOWN	0B5h	Cursor down page
Ctrl-DOWN	0B6h	Cursor down paragraph
LEFT	0B8h	Cursor left character
Shift-LEFT	0B9h	Cursor to start of line
Ctrl-LEFT	0BAh	Cursor left word
RIGHT	0BCh	Cursor right character
Shift-RIGHT	0BDh	Cursor to end of line
Ctrl-RIGHT	0BEh	Cursor right word

6.1.1 Left and Right by Character

The simple left and right movements move the cursor by one character left and right. There is no wrap around from one line to the next so if the cursor is at the extreme left or right of the video page then attempting to move it further will have no effect.

6.1.2 Start and End of Line

Moving the joystick left or right with the SHIFT key held down will move the cursor to the start or end of the current line respectively. In this context the start of the line is the first actual character in the line, discounting any spaces up to the left margin of that line. The end of the line is the last actual character in the line discounting any trailing spaces which are not represented in the buffer (although there may be spaces which are in the buffer).

6.1.3 Left and Right by Word

Joystick left or right with the CTRL key held down moves the cursor left or right by a word. The exact definition of a word is rather complex but is basically a string of alphanumeric characters and a string of non-alphanumeric characters in either order. Moving left or right by a word does wrap around between lines.

6.1.4 Up and Down by Line

The straightforward joystick up or down operation moves the cursor up or down by one line. The cursor always remains in the same column position. If the cursor is on the top or bottom line of the video page then, assuming that there are more lines in the buffer, the video page will be scrolled appropriately to bring another line onto the display. If the cursor is on the first or last line of the text then nothing will happen.

6.1.5 Up and Down by Page

Moving the joystick up with the SHIFT key held down will move the cursor up by a page. If the cursor is on the top line of the video page then the page will be scrolled one less lines than the height of the page so that the old top line is now the bottom line and the rest of the video page is new lines from the buffer. If there are insufficient lines in the buffer then the scrolling will stop when the first line of text is on the top line of the video page. If the cursor was not at the top of the page then it is moved to the top line of the page.

For moving down by a page the situation is analogous, with the cursor being moved to the bottom line of the page unless it is there already in wich case the page will be scrolled up. In either case the cursor will be left on the same column as it started on.

6.1.6 Up and Down by Paragraph

Moving up and down by paragraph (by using CTRL) is rather different from other up and down movements. In moving up by a paragraph the cursor will be put on the first character of the current paragraph, unless it is already on the first character in which case it will be put on the first character of the previous paragraph. The video page will of course be scrolled appropriately to ensure that the cursor remains on the screen.

Moving down by a paragraph always moves the cursor to the start of the next paragraph unless it is already in the last paragraph in which case it will be left at the end of that paragraph.

6.2 Inserting and Insert Mode Control (The INS key)

The insert key has three separate functions which are:

INS	0A8h	-	Insert a space
Shift-INS	0A9h	-	Insert a new line
Ctrl-INS	0AAh	-	Toggle insert/overwrite mode

6.2.1 Inserting Spaces and Lines

When the INS key is used on its own it simply inserts a space character before the cursor position and leaves the cursor on this space. It is useful for inserting a few characters while in overwrite mode (see below). This is done simply by inserting the correct number of spaces and then over-typing them with the required characters.

When the INS key is used with the SHIFT key, the current line will be split with a hard carriage return (end of paragraph marker) at the cursor position. If the cursor is at the start or end of a line then this will have the effect of inserting a blank line.

6.2.2 Toggle Insert and Overwrite Mode

When the INS key is used with the CTRL key, it toggles between insert and overwrite mode, the initial default being overwrite mode. The current mode is indicated by the cursor which is changed to a different character depending on the mode. For overwrite mode it is character number 14 (a rectangular block) and for insert mode it is character number 30 (a left pointing arrow).

In overwrite mode if a character is typed when there is already a character at the cursor position then the old character will be replaced by the new one. In insert mode the new character will be inserted before the old character and the old character along with the rest of the line will be moved one character to the right.

Whether overwrite or insert mode is selected also affects some details to do with word wrapping and splitting lines in the middle of the buffer.

6.3 Deleting and Erasing (The DEL and ERASE keys)

The DEL and ERASE keys have very similar functions, both delete characters from the buffer. Basically the DEL key goes rightwards while the ERASE key goes leftwards. Each key has three functions which correspond to the three types of horizontal cursor movement:

DEL	0A0h	Delete character right
Shift-DEL	0A1h	Delete line right
Ctrl-DEL	0A2h	Delete word right
ERASE	0A4h	Erase character left
Shift-ERASE	0A5h	Erase line left
Ctrl-ERASE	0A6h	Erase word left

6.3.1 Deleting And Erasing Characters

If used without the SHIFT or CTRL keys then DEL and ERASE each delete a single character and move the rest of the line left to fill up the gap. DEL deletes the character under the cursor leaving the cursor in the same position, while ERASE deletes the character to the left of the cursor and then moves the cursor onto the previous character.

Both functions wrap around between lines and thus can be used to join lines together. If the cursor is on the first character of a line then ERASE will join this line to the previous line. The line separator counts as one character as far as deletion goes. DEL will join the current line to the next one if it is at the end of a line.

6.3.2 Deleting and Erasing Lines

When used with SHIFT, the ERASE and DEL keys delete to the start and end of the current line respectively. If already at the start or end of the line then nothing will be done, these functions do not join lines together.

6.3.3 Deleting and Erasing Words

When DEL and ERASE are used with the CTRL key they delete one word, using the same definition of a word as cursor movement. The deletion is done by repeatedly deleting characters until the end of the word is reached. These functions will join lines together and for this purpose the line separator counts as a word. As usual DEL deletes rightwards and ERASE deletes leftwards.

6.4 The TAB key

The TAB key (key code 09h) moves the cursor to the next tab stop, or to the start of the next line if there are no more tab stops on this line. The current TAB settings can be seen on the ruler line if it is displayed.

In overwrite mode the cursor is simply moved to the next tab stop. In insert mode the next tab stop is reached by inserting spaces and moving any more text on the line to the right. When moving to the start of a new line in insert mode, spaces will be inserted up to the right margin and then a new line will be inserted (not an end of paragraph marker).

6.5 The Editing Function Keys

The more complex editing functions, and particularly the word processor type functions are carried out by using the eight function keys in conjunction with CTRL or ALT. This gives a possible 16 editing functions although only 14 of these are utilised because function key 8 is not used.

These 14 editing functions are listed here, along with their key codes and each one is then described in more detail in the following sections.

Ctrl-F1	0F0h	-	Reform paragraph
Ctrl-F2	0F1h	-	Centre line
Ctrl-F3	0F2h	-	Toggle tab
Ctrl-F4	0F3h	-	Set left margin
Ctrl-F5	0F4h	-	Release margins
Ctrl-F6	0F5h	-	Move paragraph up
Ctrl-F7	0F6h	-	Change line colour
Alt-F1	0F8h	-	Justify paragraph
Alt-F2	0F9h	-	Remove all tab stops
Alt-F3	0FAh	-	Toggle ruler line display
Alt-F4	0FBh	-	Set right margin
Alt-F5	0FCh	-	Reset margins and tabs
Alt-F6	0FDh	-	Move paragraph down
Alt-F7	0FEh	-	Change paragraph colour

6.5.1 Reform and Justify Paragraph (Ctrl-Fl and Alt-Fl)

Reform and justify paragraph are very similar functions, in fact justify does exactly the same as reform but also justifies. Both operate on the paragraph containing the cursor, and leave the cursor on the start of the next paragraph. This means that pressing one of these keys repeatedly will reform (or justify) each paragraph of a document in turn, without need for using the joystick.

Reform paragraph moves to the start of the paragraph and then walks through the paragraph to the end. As it goes it adjusts the left margin of each line to be equal to the current left margin, removes any soft spaces (left over from previous justification) and word wraps each line to the current right margin, joining lines together where possible.

The result is that the new paragraph appears exactly as it would if all the characters of the paragraph were newly typed in, so any untidy sections resulting from other editing operations will be reformed.

Justify does exactly the same as reform but it also inserts soft spaces into each line of the paragraph except the last one, to ensure that each line finishes exactly on the right margin.

6.5.2 Centre Line (Ctrl-F2)

Centre line is a fairly simple function which operates on a single line, not a whole paragraph. It inserts sufficient spaces before the line to centre it between the current margins. Leading and trailing spaces are first removed to ensure that they are not included in the centring. The cursor is left on the start of the line. If the line is too long to fit between the margins then it will be left so that it starts at the left margin position.

6.5.3 Toggle Ruler Line Display (Alt-F3)

The ruler line display is a red line which can be displayed on the very top line of the video page. It indicates where the left and right margins are set and also the positions of any tab stops.

The left margin is indicated by an "L" and the right margin by an "R". Tab stops are marked by a vertical bar and other character positions between the margins are indicated by dashes. Also if the margins are released (see below) then an asterisk will be displayed in the extreme right hand position of the ruler line.

This function key simply toggles the ruler line display on and off, the default being off. All the facilities of tab stops and margins can be used regardless of whether the ruler line is displayed or not, but it can get confusing if it is not. The built in word processor sets the ruler line display on for the user.

6.5.4 Toggle and Clear Tabs (Ctrl-F3 and Alt-F2)

The toggle tab function sets a tab stop at the current cursor column, or removes it if one was already set. Tab stops can only be set between the current margin positions, although when the margins are moved tab stops which are outside them are remembered and restored when the margins are moved back out. It is advisable to have the ruler line displayed when using this function.

All tab stops can be removed by a single function key press, including those which are outside the current margin settings (and thus not visible on the ruler line). This is useful to get rid of the standard tabs before setting up your own set.

When an editor channel is opened, the tab stops are set up by default to every eight character positions since this corresponds to the standard setting for tabs on machines with fixed tab stops.

6.5.5 Set Left and Right Margins (Ctrl-F4 and Alt-F4)

The left and right margins define what portion of the video page is used for entering and displaying text. When an editor channel is opened the margins are initialised to the widest possible setting. The user can set new margin postions by putting the cursor on the desired column and pressing the appropriate function key.

The right margin can be in any column up to two less than the video page width. Thus for a 40 column display (BASIC's default) the right margin can be any column up to 38. The left margin can be in any column from 1 up to one less than the right margin column. The default margin settings for BASIC's default channel are thus: left margin at column 1, right margin at column 38.

An attempt to set an illegal margin position will result in both margins being reset to their default settings. The applications program can use these codes to set margin positions but there is also a special function call which can be used to set the margins and also to read their current settings. This is described later.

6.5.6 Release Margins (Ctrl-F5)

As mentioned before there is a margin release function. This is in fact a toggle action, it releases margins on the first press and the re-enables them when it is pressed again. When margins are released the margins remain displayed on the ruler line and an asterisk is displayed on the extreme right hand end.

When margins are released, all operations which normally use the margin settings use the default settings instead. Thus word wrapping will occur at the last-but-two column rather than the right margin and characters may be typed in outside the margins.

6.5.7 Reset Margins and Tabs (Alt-F5)

This function key resets the margin settings to their default values and sets up the default positions of tab stops (every eight columns).

6.5.8 Move Paragraph Up and Down (Ctrl-F6 and Alt-F6)

These functions can be used to move a paragraph up or down. Each key press will move the paragraph up or down by one line, unless it is already at the start or end of the buffer. To move a paragraph by more than one line this key should be pressed repeatedly until the paragraph reaches the desired position.

The paragraph to be moved is defined to start on the current cursor line, and end at the next end of paragraph marker. Thus to move a complete paragraph the cursor should first be positioned on the first line of the paragraph. This definition of a paragraph has to be used to ensure that one paragraph can be moved through another correctly.

Although not essential it is useful to use the "colour paragraph" function before doing a series of moves up or down. This highlights the paragraph to make it easy to see what is going on, and also puts the cursor at the start so that the whole paragraph will be moved.

6.5.9 Colour Line and Paragraph (Ctrl-F7 and Alt-F7)

These functions can be used to change the colour of the text. The colour line function just affects the current line whereas the colour paragraph function affects the entire current paragraph, and has the side effect of moving the cursor to the start of the paragraph ready for paragraph moving.

Each change colour operation changes the colour to the next one of four posible colour pairs, cycling back to the first pair after the fourth. A colour pair specifies which paper and ink palette colour the the video driver will use for the text. The four colour pairs are (in order): (0,1), (2,3), (4,5), (6,7), with (0,1) being the default (normally green on black).

Note that if the editor is using a hardware text page then colour pairs (4,5) and (6,7) will in fact appear as pairs (0,1) and (2,3) since the video driver only supports two colour pairs in this mode.

7. Special Function Calls

7.1 Setting Margin Positions

The margin positions can be set by the user or the applications program by using a function key. However a more general way for the applications program to set the margin positions is provided as a special function call. The parameters for this call are:

```
Parameters:     A = Channel number (1...255)
                B = @@MARG (=24)  (Special function code)
                D = New right margin column
                E = New left margin column

Returns:        A = Status
                D = Right margin column
                E = Left margin column
```

The column numbers can be from 1 up to the width of the video page, with the usual restrictions on valid margin settings. There are two special values which can be given for the new left and right margin parameters. If either of them is 0FFh then the current cursor column will be used for that margin. If either of them is zero then that margin setting will be unchanged. Thus the current margin settings can be read without affecting them, simply by setting both D and E to zero.

7.2 Loading and Saving Document Files

Two special function calls are provided for loading and saving document files. The format of saved documents fits in with the standard EXOS file module format which is described in the EXOS kernel specification. The module type for editor document files is $$EDIT (=8) and the module header contains no other information.

The simplest of the two functions is saving. The call must specify a channel number down which the document is to be saved and this channel must be opened before making the call. The editor will write a suitable module header followed by the data of the document. It does not write out an end of file header at the end, since the application program may want to create a multi-module file.

The details of this special function call are:

```
Parameters:  A = Editor channel number (1...255)
             B = Channel number to write to (1...255)

Returns:     A = Status
```

Loading is slightly more complex. Before making the special function call, the file to load from must have been opened, and a file header with type $$EDIT must have been read in. The editor is then called, giving the channel number to load from. It first clears all the text out of the editor buffer and then loads the new document in from the file. It loads line by line and checks that each line is valid before going on to the next. If it finds an invalid line or the buffer becomes full then it stops loading and returns an error (.EDINV or .EDBUF). All previous lines in the buffer will be valid. The cursor will be left at the start of the document.

The parameters for the loading special function call are:

```
Parameters:  A = Editor channel number (1...255)
             B = Channel number to load from (1...255)

Returns:     A = Status
```

A saved document contains the characters of the text and also information about paragraph structure, the left margin position for each line, and the colour of each line. Thus if a document is saved and loaded into an editor channel with the same sized buffer and video page, then it will re-load exactly as it was before being saved.

Information about editor options and so on is not saved with the file so any tab settings, current margin settings and so on will have to be set up again when the file is loaded.

A document can be loaded into a different sized editor channel but may have to have some lines adjusted if their left margin setting is invalid. Also it may not all fit into the buffer if the new buffer is smaller.

8. Error Handling

The only error which can be generated inside the editor during normal operation (apart from loading and saving documents, opening channels, illegal special function calls, and unknown escape sequences) is .VCURS which is returned if an invalid cursor position is given in the cursor positioning escape sequence.

However, since the editor is communicating with a video page and a keyboard channel, it can get errors from these channels. Generally an error from one of these channels means that the channel is misbehaving in some way. For example if someone has re-directed the editor's video channel to a different device, or even closed it then the editor will get errors from its video channel.

If an error does come from one of these channels then the editor returns either .EKEY or .EVID error code to the applications program. It also remembers that this error occurred and whenever it is next called, with any EXOS call, it checks the channel (by writing a null to the video or reading status from the keyboard) to see if it is basically working. If it is working then it clears the error flag and carries on normally. If it is not working then it returns the appropriate error code (.EKEY or .EVID) to the user without carrying out the function call.

This procedure ensures that the applications program can tell when the editor is having trouble with its secondary channels and attempt to put things right. This is how BASIC manages to recover if the editor's video channel is closed. BASIC discovers this and re-opens it again.

Another feature is that if a NULL (ASCII code 0) is written to the editor, instead of just being ignored, it is written directly to the video page, which will ignore it. This provides a way to 'poke' the editor to check if its video channel is still there, without having any effect on the editor's data.

9. Quick Reference Summary

9.1 EXOS Calls

OPEN/CREATE CHANNEL - Treated identically. Supports multiple channels. Device name "EDITOR:". Filename and unit number ignored. EXOS variables VID_EDIT, KEY_EDIT, BUF_EDIT must be set before open.

CLOSE/DESTROY CHANNEL - Treated identically.

READ CHARACTER/BLOCK - Returns characters from buffer after allowing user to do editing. Details controlled by FLG_EDIT EXOS variable.

WRITE CHARACTER/BLOCK - Printing characters put in buffer and displayed. Responds to some control codes and ESC=. Some codes above 0A0h do editing functions.

READ STATUS - Returns C=0 if a read character call would return character immediately without allowing user a chance to edit. Returns C=1 otherwise, or C=0FFh if just finished a SEND ALL.

SET STATUS - Not supported.

SPECIAL FUNCTION - @@MARG = 24 Set and read margins.
@@CHLD = 25 Load document file.
@@CHSV = 26 Save document file.

9.2 EXOS Variables

VID_EDIT - Channel number of video page.
KEY_EDIT - Channel number of keyboard channel.
BUF_EDIT - Buffer size in multiples of 256 bytes.

FLG_EDIT - Flags to control reading from editor.
b7 - SEND NOW
b6 - SEND ALL
b5 - NO READ
b4 - NO SOFT
b3 - NO PROMPT
b2 - AUTO ERA
b0 & b1 - Not used.

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

1. Introduction

The video driver handles the display of any number of "video pages" in various different display modes, making use of most of the facilities of the NICK chip.

The display is managed in terms of video "pages", with one page corresponding to each EXOS channel which is open to the video driver. Before a channel is opened to the video driver the user must specify various parameters, such as a video mode and page size, by setting EXOS variables. A channel can then be opened to device "VIDEO:". If a filename or unit number is specified then it will be ignored. The video driver will work out how much RAM it needs for this video page and obtain that much RAM from EXOS, including enough for the various variables needed. The only limit on the number of video pages is the amount of available memory. Video pages can only use the internal 64k of video RAM.

Once the channel has been set up in this way, the user can read characters from, or write them to, the video page. The data read or written will have a different meaning for pages of different modes, particularly control characters and escape sequences.

At this stage the video page will not be visible on the screen. A special function call is required to cause a video page to be actually displayed on the screen. It is only at this time that the appropriate line-parameter blocks are set up and the text/graphics will appear. It is possible to display any vertical section of a video page at any vertical position on the screen, covering up anything which was displayed on those scan lines before. If the page width is less than the full screen width then the margins will be adjusted to display the page in the middle of the screen.

Text pages provide displaying of characters from a 128 character font which is initialised to a standard ASCII character set, but any characters may be re-defined by the user. Also text pages provide various control functions including cursor positioning, scrolling of any section of the page and automatic scrolling.

There are various different graphics modes providing a choice of resolution, number of colours and memory usage. All of the graphics modes support drawing of lines and ellipses in a variety of plotting modes and line styles (dotted lines etc.). Characters can be displayed using the same font as text pages. There is a sophisticated flood filling algorithm which will fill any arbitrary shapes subject to stack limitations.

1.1 Co-ordinate Systems

The co-ordinate system used in specifying graphic positions is standardised so that giving the same commands to two pages of different resolutions or colour modes will produce a pattern of the same size on the screen. A graphics page of full screen size will be 972 logical pixels high and 1344 pixels wide. This corresponds to twice the maximum horizontal and four times the vertical resolution available. All beam positions are specified in these co-ordinates, and depending on the colour mode the actual position will have to be an approximation. The origin for this co-ordinate system is the bottom left corner of the graphics page.

Text pages do not use this co-ordinate system, they use a system based on character positions so the top left corner is (1,1) and the top right corner (of a full screen size low resolution text page) is (1,42). Note that the origin for text co-ordinates is the top left of the page.

Attribute graphics mode pages actually keep a beam position in graphics co-ordinates and a separate cursor position in text co-ordinates. The use of these is explained later.

2. Basic Control of Video Pages

As mentioned before, each video page is a separate channel. When a channel is opened to the video driver this implies that another video page is to be created. The video driver looks at EXOS variables which specify the page size, page mode and colour mode. These variables must be set up by the user before opening a video channel. From these variables the video driver determines how much video RAM it needs and obtains that much with an EXOS function call ("Allocate channel buffer").

The video driver maintains the line parameter table in a fixed place in its absolute device RAM area. The line parameter table always consists of 28 line parameter blocks of 9 scan lines each for the display area and various other ones to generate the frame sync and borders. The first line parameter block is reserved for the status line display which is a fixed area of RAM. The other 27 line parameter blocks can display any part of any page, so display is always in vertical units of 9 pixels. All 28 line parameter blocks are initially set up to be blank (ie all border colour). The variable LP_POINTER in the EXOS variable area points to the start of the line parameter table.

2.1 Display Modes

The display mode is specified by an EXOS variable MODE_VID the allowed values of which are:

- 0 - Hardware text mode (up to 42 chars/line).
- 1 - High resolution pixel graphics.
- 2 - Software text mode (up to 84 chars/line).
- 5 - Low resolution pixel graphics.
- 15 - Attribute graphics.

Any other values will produce an error (.VMODE) when an attempt is made to open a channel. The three graphics modes correspond to the PIXEL, LPIXEL and ATTRIBUTE modes of the NICK chip (see separate NICK chip specification).

2.2 Colour Modes

As well as the display mode, each video page is of a particular colour mode. The colour mode is specified by an EXOS variable called COLR_VID. The allowed values for this variable are:

- 0 - Two colour mode
- 1 - Four colour mode
- 2 - Sixteen colour mode
- 3 - 256 colour mode

Any other values will be reduced modulo 4 and so no errors are produced. For text modes it is only useful to use two colour mode, unless the characters in the font are re-defined for doing some sort of block graphics. Also attribute mode must always be in two colour mode, although sixteen colours will actually be available on the page.

2.3 Page Size

Two EXOS variables, X_SIZ_VID any Y_SIZ_VID, define the size of the page to be created. The vertical size is specified in character rows. It can be any value from 1 to 255 although only 27 rows can be displayed on the screen at one time. The horizontal size is specified in low resolution character widths, and can be any number from 1 to 42. Invalid values will produce an error (.VSIZE) when a channel is opened.

A special function call is provided to return the size of a video page. It returns the number of lines and the number of characters per line. The number of lines will be the same as the Y_SIZ_VID EXOS variable when the channel was opened. The characters per line value returned is the actual number of characters per line so in the case of a software-text mode it will be double the value in X_SIZ_VID when the channel was opened.

The parameters for this call are:

```
Parameters:     A = Channel number (1...255)
                B = @@SIZE (=2) (Special function code)

Returns:        A = Status
                B = Number of characters per row
                C = Number of rows
                D = Mode of page (0, 1, 2, 5 or 15)
```

2.4 Display Control

Video pages are not actually displayed on the screen until the user explicitly requests this. This request is done by a special function call to the channel. The parameters for this call are:

```
Parameters:     A = Channel number (1...255)
                B = @@DISP (=1) (Special function code)
                C = 1st row in page to display (1...size)
                D = Number of rows to display  (1...27)
                E = row on screen where first row
                      should appear (1...27).

Returns:        A - Status
```

The three row parameters are all given in character row units since the area of screen specified must be a whole number of line parameter blocks. The displayed page will replace anything which was displayed on that part of the screen before. If the channel is subsequently closed then any part of the screen which was displaying that channel will be made border colour (by bringing the margins in the relevant line parameter blocks right in).

A value of 1 for the position on screen parameter (given in register E) refers to the line on the screen directly below the status line. Thus it is not possible to overlay the status line since zero will not be accepted.

If a value of zero is given for the position in the page parameter (register C) then the portion of the screen defined by the other two parameters will be blanked (ie. made entirely border colour).

If any of the parameters for the function call are invalid for any reason then an error (.VDISP) is returned.

3. Video Modes and RAM Usage

When a channel is opened the video driver obtains sufficient RAM from EXOS to support the page. This includes a certain amount for internal variables (128 bytes), an overhead of two bytes for each line of the page, and enough RAM for the display memory which will vary in size depending on the display mode and page size. Note that for any given display mode and page size, all of the four possible colour modes will use the same amount of RAM since they trade off resolution for number of colours without affecting the memory required.

There is a special function call provided which returns the actual address of the display RAM for that page. In fact two addresses are returned because some modes use two different areas of memory. The exact meaning of the addresses for each mode is described below in the section on the appropriate mode. The addresses which are returned are the addresses as seen by the NICK chip. Thus an address in the range 0000h...3FFFh corresponds to RAM segment 0FCh, 4000h...7FFFh corresponds to segment 0FDh and so on for segments 0FEh and 0FFh.

```
Parameters:     A = Channel number (1...255)
                B = @@ADDR (=3) (Special function code)

Returns:        A = Status
               BC = Main display RAM address
               DE = Secondary display RAM address
```

Note that the display RAM for a video channel can be moved by EXOS and so the addresses returned by this call will not always remain the same. The operations which can cause channel RAM areas to move are explained in the EXOS Kernel specification. The most important ones are opening and closing of other video channels and linking in user devices or resident system extensions. If any operations of this type have been performed then this special function call will have to be repeated to obtain the new addresses.

In the sections below, where HEIGHT and WIDTH are referred to in specifying RAM requirements, these values are the actual values from Y_SIZ_VID and X_SIZ_VID respectively. Thus for example a full screen size software text page has a width of 42, even though it actually has 84 characters across.

The RAM requirements given below are the amount of channel RAM which the video driver will ask for. In addition to this each page will require a channel descriptor (of 16 bytes) which is allocated by EXOS.

3.1 Hardware Text Mode (Mode 0)

In hardware text mode, one byte of RAM is allocated for each character position on the page. These bytes contain the character codes for the characters displayed on the page and the NICK chip itself translates these into character shapes using the font, when the display is generated. If the top bit of the character code in the display RAM is set then this character will be displayed using palette colours 2 and 3 rather than 0 and 1.

Initially the ASCII map starts with the top left character of the page and continues across the first line followed by the second line and so on down the page. However once any scrolling operations have been performed the ordering of lines on the page will be different so the first byte of the ASCII map will be the first character of a line but not necessarily the top line. The lines can end up in any arbitrary order and even clearing the page will not re-order them.

The RAM usage and address parameters for a page of this mode are:

DE = BC = Start of ASCII character map

Total RAM = 128 + 2*HEIGHT + WIDTH*HEIGHT

3.2 Software Text Mode (Mode 2)

Software text mode maintains an ASCII copy of the page which corresponds to the memory used in hardware text mode. This is one byte for each character on the page. In addition to this it has a complete bit map of the page. The video driver itself builds up the character shapes in this bit map from the character font. This bit map is used by the NICK chip to generate the display, the ASCII map is only used internally by the video driver software.

The bit map initially corresponds directly with what is seen on the screen (assuming the video page is displayed), with one bit corresponding to each pixel. The first byte therefore corresponds to the first eight pixels on the top line of the page, which is the top line of the first character. The next byte corresponds to the top line of the next character and so on until the end of the first row of characters. The next byte will correspond to the second scan line of the first character. This continues for nine scan lines to complete the first row of characters. Subsequent rows of characters are built up in the same way.

The same comments about scrolling apply to software text pages as to hardware text. Scrolling operations can re-order the lines of a page in any arbitrary order. The ASCII map and the bit map are always re-ordered in the same way.

When the video driver is putting characters from the font onto a software text page it masks out the top and bottom bits of each byte and inserts colour information into these bits in the bit map. The values of these two bits control which palette colours are used to display this byte (all 9 bytes in a character will be the same colour). The meaning of these bits is:

bit-0	bit-7	palette colours used
0	0	0 and 1
0	1	2 and 3
1	0	4 and 5
1	1	6 and 7

The RAM requirement and address parameters for a software text page are:

BC = Address of start of bit map (top scan line of top left character).
DE = Address of start of ASCII map (top left character).

Total RAM = 128 + 2*HEIGHT + 20*HEIGHT*WIDTH

3.3 Pixel Graphics Modes (Modes 1 and 5)

The two pixel graphics modes are high resolution (mode 1) and low resolution (mode 5). The only difference between these modes is the amount of RAM they use, and therefore the resolution. A pixel graphics page has an area of RAM which is a straightforward bit map of the screen. The first byte corresponds to the top left of the page, the next byte to the second byte on the top scan line and so on to the end of the first scan line. This is then repeated for the next scan line and so on until the bottom of the page.

81

The mapping of these bytes into pixels depends on the colour mode and is described in the separate NICK chip specification.

In low resolution pixel mode the full screen width is 42 bytes and in high resolution pixel mode it is 84 bytes. High resolution pixel mode thus uses twice the amount of RAM to cover the same screen area as low resolution pixel mode.

The address parameters and RAM usage for the two pixel graphics modes are:

```
BC = Address of top left byte of display RAM
DE = Value is un-defined

Total RAM (low resolution)  = 128 + 2*HEIGHT + 9*WIDTH*HEIGHT
Total RAM (high resolution) = 128 + 2*HEIGHT + 18*WIDTH*HEIGHT
```

3.4 Attribute Graphics Mode (Mode 15)

An attribute graphics page requires two areas of RAM of equal size. The first of these (the pixel data area) is a bit map of the video page which corresponds exactly to the bit map for a two colour low resolution pixel graphics page, with each byte defining eight pixels. The second RAM area is the attribute data. Each byte in the attribute data area defines two palette colours in the range 0...15, the INK attribute and the PAPER attribute. The eight bits in the corresponding pixel data byte define which of the two colurs each of the eight pixels covered by this byte will be.

The format of a byte in the attribute data area is:

```
b7:b6:b5:b4 - PAPER colour, used if a bit in pixel
                     data byte is clear.
b3:b2:b1:b0 - INK colour, used if bit in pixel data
                     byte is set.
```

The address parameters and RAM requirements for an attribute graphics page are:

```
BC = Address of start of pixel data area
DE = Address of start of attribute data area

Total RAM = 128 + 2*HEIGHT + 18*WIDTH*HEIGHT
```

4. Character Output

The screen driver supports both the single character write and the block write EXOS function calls. A block write is exactly equivalent to writing all the characters individually, except that it is rather faster as it avoids the overhead of going through EXOS for every character.

4.1 Printing Characters

All character codes above 31 will be treated as printing characters and will be put at the appropriate place on the video page. All modes have some sort of "cursor" which moves when a character is printed but the details vary between different modes.

The bit maps for characters are stored in a fixed character font which is initialised to an ASCII character set. Each character is eight bits wide and nine bytes deep, including the space between characters and between lines. The user can re-define any of these characters by an escape sequence as specified below.

Character codes in the range 32 to 127 will be displayed as the correct character number from the font. Characters in the range 128 to 255 will be displayed as characters 0 to 127 from the font. Thus writing character 160 (128+32) to a video page will have exactly the same effect as writing character 32. However writing character 159 (128+31) will display character number 31 from the font on the page but writing character 31 will do nothing because it will be interpreted as a control code (this particular control code is ignored).

4.1.1 Text Mode Character Printing

Text pages (modes 0 and 2) maintain a single text cursor which is in text co-ordinates. The printing character is displayed at this position and the cursor moved to the next character slot. At the end of a line the cursor automatically moves to the start of the next line, with automatic scrolling if it is at the bottom of the screen (this automatic scrolling can be disabled).

4.1.2 Pixel Graphics Mode Character Printing

Pixel graphics pages maintain a beam pointer in graphics co-ordinates. The printing character is displayedat this beam position and the beam moved to the next character position, moving to the start of the next line if at the end of a line. Characters can be displayed at any pixel position, not just on character boundaries. There is no scrolling. If the beam is too near the bottom of the page to fit the whole character on then it will not display anything.

Characters are displayed in the current ink colour regardless of whether the beam is on or off. The character is displayed by reading bits out of the font and if the bit is set then a pixel is plotted in the current ink colour using the current line mode. If the bit is clear then the corresponding pixel will be left unchanged. A character will therefore overlay rather than replace anything which is already on the page.

The characters will be the correct shape in any colour mode but the aspect ratio will vary with different modes. To improve legibility the character height is doubled for sixteen and 256 colour mode.

4.1.3 Attribute Graphics Mode Character Printing

An attribute graphics page maintains a text cursor in text co-ordinates and a separate graphics beam position in graphics co-ordinates. Characters are printed at the text cursor position and so will always be in exact character positions. At the end of a line the text cursor will go on to the start of the next line and at the end of the page it will go back to the top left of the page - there is no scrolling.

How the character is displayed depends on the current value of the attribute flags byte for this page. This is a byte which can be set by an escape sequence and is described in more detail later on. It basically controls what sections of the attribute and pixel data will be affected by writing characters or plotting graphics.

4.2 Control Codes and Escape Sequences.

Character in the range 0 to 31 are control characters and are not printed. Some of these are interpreted by video pages, depending on the mode. Any which are not understood are simply ignored. A special control code is ESCAPE (ASCII 1Bh) which is used to start an escape sequence for carrying out various functions.

Below is a list of the control codes and escape sequences interpreted by the various modes. The more complex of these are explained in the next section.

4.2.1 Codes Interpreted by Any Video Page

^Z (1Ah) - Clear entire page and home cursor/beam.

^J (0Ah) - Line-feed. Move cursor down to next line (scrolls if at bottom of screen in text mode and scroll is enabled.)

^M (0Dh) - Carriage return. Returns cursor to start of current line

^^ (1Eh) - Cursor/beam home. (ASCII RS)

escK - Define character (see below)

escC - Set all palette colours \ see below for
escc - Set one palette colour / parameters.

escI<n> - Set ink colour to <n> \ See below for details
escP<n> - Set paper colour to <n> / in different modes

esc=<y><x> - Set cursor position (see below)

4.2.2 Codes Interpreted by Text Pages Only

^Y (19h) - Clear to end of line. Does not move cursor.

^H (08h) - Cursor left. (ASCII BS)
^I (09h) - Cursor right. (ASCII TAB)
^K (0Bh) - Cursor up. (ASCII VT)
^V (16h) - Cursor down. (ASCII SYN)

esc? - Read cursor position. Also supported in attribute mode. (see below)

esc.<n> - Set cursor character to character code <n>.

escM<n> - Set cursor to palette colour <n>

escO - Set cursor display on.
esco - Set cursor display off.

escS - Set automatic scroll on
escs - Set automatic scroll off

escU<m><n> - Scroll up lines (m-20h) to (n-20h) m <= n
escD<m><n> - Scroll down lines (m-20h) to (n-20h) m <= n

4.2.3 Codes Interpreted by Graphics Pages Only

escA<xx><yy>	-	Position beam at co-ordinates (xx,yy) where xx & yy are each 16-bit hex numbers specified low byte first.
escR<xx><yy>	-	Relative beam movement by amount (xx,yy).
esc@	-	Read beam position. (see below)
escS	-	Set beam on.
escs	-	Set beam off.
esc.<n>	-	Set beam to line style <n> - see below.
escM<n>	-	Set beam to line mode <n> - see below.
esca<n>	-	Set attribute flags byte to <n>. Only allowed in attribute mode (see below).
escF	-	Graphics fill - see below.
escE	-	Plot ellipse - see below.

5. Details of Escape Sequences

5.1 Position Cursor

The escape sequence to position the cursor works in all modes. In text mode it simply moves the cursor. In attribute graphics mode it moves the text cursor but leaves the graphics beam pointer alone. In pixel graphics modes it moves the graphics beam pointer to the appropriate text co-ordinates, so it will be on a character boundary.

The format of the escape sequence is:

esc=<y><x>

This sets the cursor to row (y-20h) and column (x-20h). If either <x> or <y> is 20h (thus setting row or column zero) then that co-ordinate will remain un-changed. This allows just the row or column to be set.

5.2 Define Character

This escape sequence allows the user to re-define one of the 256 characters. Although it is sent to a specific channel, it actually affects the global character font and will thus affect other channels. Characters already displayed on a page will only be affected for hardware text pages. In other modes only subsequently written characters will be affected. The syntax of the escape sequence is:

escK<n><r1><r2><r3><r4><r5><r6><r7><r8><r9>

where: <n> is the character number (0...255)
<r1>...<r9> are the bytes for the nine rows of the character. <r1> is the top row.

Note: In high resolution text mode, only the middle six bits of the character bytes will actually be displayed as the other two are masked out and used to control the colour selection.

5.3 Palette Colours

Each video page has a palette of eight colours associated with it which is initialised to a useful set of colours when the channel is opened. There is an escape sequence with which the user can change all these colours. the format of this is:

escC<c><c><c><c><c><c><c><c>

Each <c> is a byte specifying one of the palette colours and there must always be eight of them.

There is another escape sequence which allows just one palette colour to be changed. The format of this is:

escc<n><c>

Where <n> is the palette colour number 0...7 and <c> is the new value for this palette colour.

When new palette colours are selected any line parameter blocks which correspond to this video page will be updated so the colours on the screen will change.

5.4 Ink and Paper Colours

The user may specify a palette colour for both the ink and paper colour with separate escape sequences. For all video modes the ink colour defaults to one and the paper colour to zero.

For pixel graphics pages the allowed ink and paper colours depend on the colour mode, so it is 0 or 1 in two colour mode, 0...3 in four colour mode, 0...15 in sixteen colour mode and 0...255 in 256 colour mode. Pixels are always plotted in the current ink colour. The paper colour of the display is only changed when the page is cleared.

For attribute graphics mode the ink and paper colours can be in the range 0...15 and they control what colours are put into the attribute bytes. See also the section on the attribute flags byte below which determines whether the attribute and pixel data is actually updated.

In four, sixteen or 256 colour text modes the ink and paper colours have no useful effect because the palette colours for each pixel are determined directly from bits in the character font. In fact they do have some interaction with the displayed colours and it is best to leave them set to their default values. These modes are only useful if the characters have been redefined to provide some sort of block graphics.

In two colour hardware text mode the ink and paper colours are always (0,1) or (2,3). If the ink or paper is changed then the other one is changed to correspond.

In two colour software text mode four colour pairs are available, (1,0), (2,3), (4,5) and (6,7). Characters are always printed in the current colour pair.

5.5 Graphics Line Style

For a graphics page the line-style may be set with an appropriate escape sequence. The line style affects line drawing and ellipse drawing but not character plotting or filling. The values for the line style byte are:

```
      1    -   Solid line   (default)
   2..14   -   Various types of broken and dotted lines.
```

5.6 Graphics Line Mode

An escape sequence specifies the line mode byte which has the following meanings:

```
   0   -   PUT plotting  (default)
   1   -   OR plotting
   2   -   AND plotting
   3   -   XOR plotting
```

For pixel graphics pages when plotting a pixel the current ink colour is combined with the old colour of the pixel according to the operation selected by the line mode and then stored.

5.7 Attribute Flags Byte

The attribute flags byte is only supported for attribute graphics pages. It consists of eight separate flags, four for plotting graphics (points, lines, ellipses and filling) and four for plotting characters. The basic operation which is affected is that of plotting a pixel.

When plotting a pixel in attribute mode there are three items which can be affected. There is the bit in the pixel data byte which corresponds to this pixel, and there are the two colours (ink and paper) in the attribute byte which corresponds to this pixel data byte. The attribute flags byte contains a flag for each of these three items which controls whether or not it is affected when a pixel is plotted. If the ink attribute is to be affected then it will be set to the current ink colour. If the paper attribute is to be affected then it will be set to the current paper colour.

The pixel data is rather more complex. Assuming that the pixel data is to be affected then there is another bit in the attribute flags byte which controls what is done with the pixel data, in conjunction with the current line mode. For plotting characters, if this bit is set then the character will be complemented before being plotted. For plotting graphics, if the corresponding bit is set then a zero will be put into the pixel bit instead of the usual one. When plotting graphics (but not characters) the bit is not in fact put directly into the pixel byte but is combined with the current value of the bit according to the current line mode. This means for example that exclusive or plotting will still work.

The top four bits of the attribute flags byte control character plotting and the bottom four control graphics plotting. For all bits the 'normal' state is zero which results in all attributes and pixel data being affected, and plotting to be in the normal sense. The assignement of bits is:

bit-7	Affect paper attributes in character plotting
bit-6	Affect ink attributes in character plotting
bit-5	Affect pixel dtaa in character plotting
bit-4	set to complement character before plotting
bit-3	Affect paper attributes in graphics plotting
bit-2	Affect ink attributes in graphics plotting
bit-1	Affect pixel data in graphics plotting
bit-0	set to do graphics plotting in 0's instead of 1's

Note that when filling bit-1 of the attribute byte is assumed to be clear regardless of its actual state. This is necessary because if the pixel data were not affected then the fill algorithm would never terminate.

5.8 Graphics Fill - Paint

The graphics fill command is a simple escape sequence which does a fill from the current beam position. It fills in the current ink colour up to any boundary which is not the same colour as the current beam position. It handles concave shapes and tests for reaching the edge of the video page. It may fail to fill the entire shape if it runs out of stack but this should only happen with extremely complex shapes since it does garbage collection on the stack when it gets full.

5.9 Graphics Ellipse Drawing

The ellipse drawing routine takes two 16-bit parameters (low byte first) specifying the x and y radii. To draw a circle these should be the same value. The centre of the ellipse will be at the current beam position. The format of the escape sequence is:

escE<xx><yy>

6. Character Input

6.1 Simple character input

When a character is read from the video driver the result depends on the mode. In text modes the ASCII code of the character at the cursor position will be returned, without moving the cursor. In graphics mode the palette colour of the pixel at the current beam position will be returned. This will be 0 or 1 in two colour mode, 0...3 in four colour mode, 1...15 in sixteen colour or attribute modes and 0...255 in 256 colour mode.

6.2 Reading Cursor Position

The escape sequence: esc? is supported in text and attribute graphics modes. It triggers the video channel to return the current cursor co-ordinates as the next two characters read from this channel. The co-ordinates will be returned in the same way as they are specified for cursor positioning, ie. with a 20h offset added on.

6.3 Reading beam position

This is supported by graphics pages only (pixel and attribute modes). The escape sequence: esc@ will trigger the channel to return the current graphics beam position as the next four bytes read from the video page. The co-ordinates are returned in the same format as they would be specified for an absolute beam position.

7. Miscellaneous

7.1 Status Line

As mentioned earlier there is a special status line display which is outide the normal page/channel structure. The first line parameter block is reserved for this status line and will never be used to display any video page. The two byte variable LP_POINTER which is at a fixed address in the EXOS variable area contains the address of the start of this line parameter block. This is used by the cassette driver to modify the palette colours of the status line to provide the cassette level meter display. It could also be used by other programs to manipulate the status line display.

An EXOS variable (ST_FLAG) is provided which will cause the status line to be displayed if it is zero and removed from the display if it is non-zero (that region of the screen will be border colour). This is implemented by the video driver examining ST_FLAG every interrupt and setting the margins in the reserved line parameter block appropriately.

There is a two byte pointer (ST_POINTER) defined at a fixed address in EXOS variable area which contains the address of the 40 byte area of RAM which is the status line. This pointer will point to Z-80 page-2 and the RAM will be at this address in segment 0FFh. The user or external devices can access the status line by finding its address from ST_POINTER. Built in devices can access it directly by using the public sysmbol ST_LINE which is the start of the status line RAM (in Z-80 page-2 segment 0FFh).

7.2 Border and Fixed Bias Colours

Two EXOS variables (BORD_VID and BIAS_VID) are provided to control the hardware border and fixed colour bias registers. The values in these variables are written out to the NICK chip on every video interrupt. The border colour is written directly to the border register. The top 5 bits of the fixed bias variable are written to the bottom 5 bits of the fixed bias register, and the top bit of the register is set according to the EROS variable MUTE_SND since it is used to silence the internal speaker.

7.3 Resetting the Character Font

As mentioned before the video driver maintains a single 128 character font which is used for all text pages and also for displaying characters on graphics pages. This is initially set up at cold start time, but is not re-initialised at subsequent device initialisations. Thus if any characters in the font are re-defined then these re-definitions will survive a warm reset or a transfer of control to a new applications program.

The font can be reset to its initial state by making a special function call to any video channel. This will set all 128 characters back to their initial shapes. Note that this will affect all characters on a hardware text page instantly but will only affect subsequently printed characters on a software text or graphics page. The parameters for this call are:

```
Parameters:     A = Channel number (1...255)
                B = @@FONT (=4) (Special function code)

Returns:        A = Status
```

8. Quick Reference Summary

8.1 EXOS calls.

OPEN/CREATE CHANNEL - Treated identically. Supports multiple channels. Device name "VIDEO:". Filename and unit number ignored. EXOS variables MODE_VID, COLR_VID, X_SIZ_VID, Y_SIZ_VID must be set before open.

CLOSE/DESTROY CHANNEL - Treated identically.

READ CHARACTER/BLOCK - Returns cursor character or pixel colour. Can be used to read cursor/beam position.

WRITE CHARACTER/BLOCK - Displays printing characters. Many control codes and escape sequences interpreted to provide special features.

READ STATUS - Always returns C=0.

SET STATUS - Not supported.

SPECIAL FUNCTION - @@DISP = 1 Display page on screen
@@SIZE = 2 Return mode & size of page
@@ADDR = 3 Return video RAM address
@@FONT = 4 Reset character font

8.2 EXOS Variables

```
MODE_VID   - Video mode                  \
COLR_VID   - Colour mode                  \  Must be set up before
X_SIZ_VID  - Characters/line (1...42)     /  a channel is opened
Y_SIZ_VID  - Lines in page               /

ST_FLAG    - Zero to display status line.
BORD_VID   - Overall screen border colour
BIAS_VID   - Colour bias for palette colours 8 to 15.
```

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

93

PROGRAMMING THE NICK CHIP

In contrast to the majority of video display devices which allow the user various modes for the whole display, NICK allows many different modes in the same display frame.

Principal features of NICK are:-

* Mixed mode displays
* User definable characters from fonts of 64, 128 and 256
* 8-bit colour output (256 colours)
* 2,4,16, and 256 colours per line chosen from 256
* Maximum resolution (using interlace) 672 * 512
* Cell based graphics, bitmap and characters
* Characters any height from 1 to 256 scanlines
* Choice of 256 border colours
* User defined screen width and height
* External colour input (unlimited sprites or TV camera)
* Efficient use of RAM (can work with < 1K of RAM)
* 4 colour-pairs in 84 column tet mode
* Special use of bits to increase colour options
* Pointer-based memory mapping, for flexibility and speed

  95

CONTROL REGISTERS

****

Input-output addresses 080H to 08FH are reserved for NICK registers although only 080H to 083H are used at present. These registers are write-only.

080H	/FIXBIAS		
	d7		VC1 output used to kill external colour
	(d6,d5)		(PRIOR1,PRIOR0) external colour priority,working in conjunction with an external 4-bit (16 colour-value) input on EC0-EC3.
		=00	ECO-EC3 select corresponding palette colour whenever the display is active and /EXTC is low.
		=01	The external colour on ECO-EC3 selects the corresponding palette colour if /EXTC is low and the internal display is generating a palette colour in the range COL8-COL15.
		=10	The external colour on ECO-EC3 selects the corresponding palette colour if /EXTC is low and the internal display is generating a palette colour in the range COL8-COL15 OR the external colour is in the range COL0-COL7 (EC3 low).
		=11	The external colour on ECO-EC3 selects the corresponding palette colour if /EXTC is low and the internal display is generating a palette colour in the range COL8-COL15 OR the external colour is in the ranges COL0-COL3 or COL8 - COL11 (EC2 low).
	(d4..d0)		colour bias for palette colours 8-15.
081H	/BORDER		
	(d7..d0)		8 bit border colour
082H	/LPL		
	(d7..d0)		(all ..a4) of pointer to line parameter table in video RAM. The index into an entry of 16 bytes, (a3..a0) is generated by the hardware.
083H	/LPH		
	d7		/(load line parameter base) normally 1
	d6		/(clock in line parameter base)
	(d3..d0)		(al5..al2) or pointer to the line parameter table in video RAM.

The video display is controlled by values loaded into the video RAM segments (up to 64K at the top of the 4M space) by the Z80. Once this 'line parameter table' has been loaded in and the line parameter base register has been loaded the display requires no more action on the part of the Z80.

The visible display is split up into 'video mode lines'. These 'modelines' are made up of 1 to 256 scanlines. (A scanline is one scan of the electron beam across the CRT and takes about 64 microseconds.)

The following 16 registers are loaded from the line parameter table before each modeline:

SC	scanlines in this modeline (two's complement)
MB	the MODEBYTE (defines video display mode)
LM	left display margin etc.
RM	right margin etc.
LD1L	(a7..a0) of line data pointer LD1
LD1H	(a8..a15) of line data pointer LD1
LD2L	(a7..a0) of line data pointer LD2
LD2H	(a8..a15) of line data pointer LD2
COL0	8 bit value of palette colour £0
COL1	" £1
COL2	" £2
COL3	" £3
COL4	" £4
COL5	" £5
COL6	" £6
COL7	" £7

BUS ACTIVITY FOR THE VARIOUS MODES

The possible video modes are:

VSYNC no border colour and use margin information to control positioning of the vertical sync pulse. This gives considerable interlace flexibility.

PIXEL use information pointed to by LD1 as a bit mapped display.

ATTR use information pointed to by LD2 as a 2-C bitmap display and information pointed to be LD1 as cell-based graphics attributes (ie: to define paper and ink colours in the cell).

CH256 use information pointed to by LD1 as indices of characters pointed to by LD2. These characters can be any number of lines deep (up to 256).
NB: offsets in the font pointer define which line of the character to start on.

CH128 As above but assumes a font of 128 characters.

CH64 As above but assumes a font of 64 characters.

LPIXEL As for pixel mode but with half the horizontal resolution.

Note that all these modes may be mixed on the same screen and that one has the choice of 2-C, 4-C, 16-C and 256-C colour modes for the PIXEL, LPIXEL, CH256, CH128 and CH64 modes. Also note the special interpretation of certain bits of display data described below.

Details of bus use during a memory cycle in the various modes:

PIXEL	/VDC1	/VDC2
Address	LD1(15....0)	LD1(15....0)
Data into	BUF1(7....0)	BUF2(7....0)

BUF1 and BUF2 are loaded sequentially into the shift register and clocked out MSB first, ie. both are display bytes. The line data pointer LD1 is incremented twice in each memory cycle. Screen data is fetched from memory and the LD1 counter is incremented only in the active part of the display, ie. between the left and right margins of a scanline. The scanline count loaded at the beginning of the PIXEL modeline determines how many scanlines in this ~~scanline~~ MODE.

[illegible]	/VDC1	/VDC2
ess	LD1(15....0)	LD2(15....0)
a into	BUF1(7....0)	BUF2(7....0)

Cell based (parallel attribute) graphics. LD1 is used as a pointer to the colour array (paper and ink colours) and LD2 points at 2-C pixel data, ie. the display bytes. LD2 is incremented once every memory cycle while the display is active and keeps incrementing up for all scanlines in the modeline. LD1 restarts from the same address for each scanline so attribute data in BUF1 applies to cells which are 8 bits wide and have a depth of the number of scanlines in the modeline.

The character modes involve indirection through the character font. A different font may be define for each modeline and line by line vertical scrolling is obtained by offsetting the original index.

56	/VDC1	/VDC2
ess	LD1(15....0)	LD2(7....0), BUF1(7....0)
a into	BUF1(7....0)	BUF2(7....0)

LD1 is reloaded at the start of each scanline and acts as a pointer into a section of RAM containing the indices of the characters to be displayed. It is incremented once in each memory cycle. LD2 is a pointer into the character font to be used and is incremented at the start of each scanline (it points to a row of a character in the font). Thus the font consists of 256 bytes defining the first row of each character and then another 256 bytes for the next row of each character etc. If the characters are 9 lines deep this requires 2304 bytes of character font (256*9). The data in BUF2 is loaded into the shift register.

93

	/VDC1	/VDC2	
CH128			
Address	LD1(15....0)	LD2(8....0)	BUF1(6....0)
Data into	BUF1(7....0)	BUF2(7....0)	

This is basically the same as the 256 character font mode but note that the font for ~~64~~ 128 9 line deep characters only requires 1152 bytes of memory.

	/VDC1	/VDC2	
CH164			
Address	LD1(15....0)	LD2(9....0)	BUF1(5....0)
Data into	BUF1(7....0)	BUF2(7....0)	

This is basically the same as the 256 character font mode but note that the font for 64 9 line deep characters only requires 576 bytes of memory.

	/VDC1	/VDC2
LPIXEL		
Address	LD1(15....0)	LD1(15....0)
Data into	BUF1(7....0)	BUF2(7....0)

This is much the same as the PIXEL mode except that the LD1 pointer is only incremented once in each memory cycle and the BUF2 data is not used. This gives half the horizontal resolution of the PIXEL mode.

VSYNC

No use is made of the information loaded from memory. It is equivalent to the LPIXEL mode.

THE LINE PARAMETER REGISTERS

****

SC — This is a two's complemented count of the number of scanlines in the modeline, ie: OFFH for one scanline in modeline.

MB d7 — If set this takes the VIRQ interrupt line low for the duration of the modeline. In conjunction with the DAVE chip this causes an interrupt to be generated at the beginning of the modeline.

(d6,d5) — Defines the colour mode:

00 2-C Two colour mode. If a bit in the byte of display data is 1 a pixel of palette colour £ 1 is output and if 0 a pixel of palette colour £0. The bits are output to the screen in the following order:

```
---------------------------------------
: d7 : d6 : d5 : d4 : d3 : d2 : d1 : d0 :
---------------------------------------
```

01 4-C Four colour mode. Pairs of bits in the byte of display data define the colour of the pixel displayed. 00 for palette colour £0, 01 for palette colour £1, 10 for palette colour £2, and 11 for palette colour £3. The pixels are displayed in the following order:-

```
-------------------------------------
:(d7,d3) : (d6,d2) : (d5,d1) : (d4,d0):
-------------------------------------
```

10 16-C Sixteen colour mode. Groups of 4 bits in the byte of display date define the colour of the pixel displayed. 0000 for palette colour £0 up to 1111 for palette colour £15. The pixels are displayed in the following order:-

```
-------------------------------------
:  (d7,d3,d5,d1)   :   (d6,d2,d4,d0)  :
-------------------------------------
```


Note that palette colours £0 to £7 have 8-bit values loaded from the line parameter table at the start of each scanline but that palette colours £8 to £15 have 8-bit values defined as follows:-

palette colour £8 = (f4,f3,f2,f1,f0,0,0,0)
palette colour £9 = (f4,f3,f2,f1,f0,0,0,1)
......
palette colour£15 = (f4,f3,f2,f1,f0,1,1,1)

where (f4,f3,f2,f1,f0) are the low 5 bits of the FIXBIAS register.

11 256-C Two hundred and fifty six colour mode. In this mode the byte of display data defines the colour of a single display pixel.

The actual colour produced is as follows:-

```
RED   = [b0]*(4/7) + [b3]*(2/7) +  [b6]*(1/7)
GREEN = [b1]*(4/7) + [b4]*(2/7) + [b7]*(1/7)
BLUE  = [b2]*(2/3) + [b5]*(1/3)
```

(Note that this specification of red, green and blue also occurs when allocating a colour to the palette.)

d4 =0 for /VRES. The VRES mode the LD1 and LD2 data pointers are reloaded at the start of each scanline' and so the same display pattern is repeated for each scanline of the modeline.

(d3,d2,d1) defines the video display mode (see above):

000	VSYNC mode
001	PIXEL mode
010	ATTR mode
011	CH256 mode
100	CH128 mode
101	CH64 mode
110	unused at present
111	LPIXEL mode

d0 If 1 this forces a reload of the line parameter base register. This will normally be programmed to occur at the end of each video frame.

LM d7 =1 for MSBALT, ie: if the top bit of the display byte is 1 this causes palette colours £2 and £3 to be selected instead of £0 and £1 in the 2-C display mode. If the top bit is 0 palette colours £0 and £1 are used as usual. In both cases the top bit seen by the shift register is forced to 0. This mode is useful in simulating an 80 coloumn VDU in the PIXEL mode. Since the msb or LHS of any character is 0 for character spacing it can be used to highlight areas of text.

d6 =1 for LSBALT ie: if the bottom bit of the display byte is 1 this causes palette colours £4 and £5 to be selected instead of £0 and £1 in the 2-C display mode. If the top bit is 0 palette colours £0 and £1 are used as usual. In both cases the top bit seen by the shift register is forced to 0. This mode is useful in simulating an 80 column VDU in the PIXEL mode, Since the lsb or RHS of any charater is 0 for character spacing it can be used to highlight areas of text.

(d5...d) define the left hand margin of the active display. In practice this value will not be below 10 for the left hand edge of the CRT. The display changes from being border colour at the left hand margin and the display data counters start being incremented. The left hand margin defines the start of the vertical sync pulse in the VSYNC video mode.

RM d7 ALTIND1 If a 2-C character mode is selected this will cause characters with an index above 080H to have a paper of palette colour £2 and an ink of palette colour £3 instead of £0 and £1.

d6 ALTINDO If a 2-C character mode is selected this will cause characters which have their next to most significant bit set to swap palette colours as follows:-

£0 - > £4
£1 - > £5
£2 - > £6
£3 - > £7

103

(d5...d0) These bits define the right hand side of the active display. The maximum value is normally 54 for the right hand edge of the CRT. The display returns to the border colour at the right hand margin and the line data pointers are not incremented until the next left hand margin. In the VSYNC video mode the right hand margin defines the end of the vertical sync pulse.

LD1L This 8-bit value defines the starting value of (a7..a0) of the line data pointer LD1.

LD1H This 8-bit value defines the starting value of (a8..a15) of the line data pointer LD1.

LD1 is used as a pointer to the next byte of display data in the PIXEL and LPIXEL modes. In the CH256, CH128 and CH64 modes it is the index of a character in the character font. In the ATTR mode it points to attribute information.

LD2L This 8-bit value defines the starting value of (a7..a0) of the line data pointer LD2.

LD2H This 8-bit value defines the starting value of (a8..a15) of the line data pointer LD2.

LD2 is used as a pointer for pixel information in the ATTR display mode and as a pointer to the character font in the CG256, CH128 and CH64 modes. It is not used in the PIXEL and LPIXEL modes.

COL0 Palette colour £0. This is the paper colour in 2C modes though note the exceptions above.

COL1 Palette colour £1. This is the ink colour in 2C modes though note the exceptions above.

COL2 Palette colour £2 (Alternate paper)

COL3 Palette colour £3 (Alternate ink)

COL4 Palette colour £4

COL5 Palette colour £5

COL6 Palette colour £6

COL7 Palette colour £7

********************* END OF DOCUMENT *********************

1. Introduction

The sound device driver provides all the sound control in the machine. It provides an interface which allows the user to manipulate most features of the sound chip.

It only allows one EXOS channel to be open to it at a time. The sound chip has four sound sources - three tone channels and a noise channel. The sound driver maintains a queue of sounds for each of these sound channels. Each sound in the queue will be played in turn when the one before it is finished. These sound queues are of a fixed maximum size of 25 sounds in each queue and will be held in the channel RAM.

As well as the sound queues, the sound device maintains a list of envelopes each with an envelope number 0...254. Each sound in the queues refers to a specific one of the envelopes or to the "null" envelope 255. When the sound is actually played the specified envelope controls changes in pitch and left and right amplitude of the sound throughout its duration. The storage for envelopes is in the sound device's channel RAM and the size of it must be specified with an EXOS variable before opening the channel.

2. General Device Interface

The sound device is write only - it will not accept any read function calls. Printing characters are ignored. All the sound functions are controlled by various control codes and escape sequences. It should accept write character and write block function calls in the obvious way.

It will have an interrupt routine which is entered 50 times per second (once every TV frame). This scans the sound queues and processes any sounds which are waiting or are currently being played. Each 20ms period is called a 'TICK' and all timing is in terms of ticks.

There are no special function calls for the sound driver. The following EXOS calls produce a FUNCTION NOT SUPPORTED error:

READ CHARACTER
READ BLOCK
SET/RETURN CHANNEL STATUS
SPECIAL FUNCTION

An EXOS variable BUF_SND is used to specify how much storage is required in channel RAM for envelopes and must be set up before a channel to the sound driver is opened. It is specified in phases and the sound driver will obtain enough RAM to guarantee that the user can define envelopes with a total of the requested number of phases. Thus if the user requests 20 phases then he will be able to define one envelope with 20 phases in it or 20 envelopes each of one phase. Because of the overhead associated with each envelope, the former case takes up rather less RAM than the latter, so if 20 phases are requested then probably more than that number can be defined before storage will be exhausted since most envelopes have more than one phase. The number of phases requested can be 2...255.

3. Envelopes

3.1 General Description of Envelopes

An envelope consists of a series of between 1 and 40 phases each of which will be executed in turn when the envelope is used. Each phase is defined by four numbers:

PD - Duration of this phase in ticks (16 bits).
CP - Change value for pitch in 1/512 semitones (16 bits).
CL - Change value for left amplitude (8 bits).
CR - Change value for right amplitude (8 bits).

The change values are signed numbers to allow a change in either direction, up or down in pitch and louder or softer in amplitude. The pitch change can be any signed 16-bit number -32768...32767. The amplitude change values must be in the range -63...+63. They specify the total change in the appropriate parameter which is required during this phase. The specified change in pitch or amplitude will be spread linearly over the specified number of ticks as will be described later.

One particular phase of the envelope may be distinguished as the start of the release phase. The effect of this will be described in detail later but basically controls the dying away of the sound after the note has really finished.

3.2 Format of Envelope Definition

Envelopes are defined by an escape sequence sent to the sound channel:

```
esc E <en> <ep> <er> [ <cp> <cl> <cr> <pd> ]*
```

<en> = Envelope number 0...254 (8 bits).

<ep> = Total number of phases in envelope 1...40 (8 bits).

<er> = Number of phases before release (8 bits). If no release phase is required then this should be 0FFh which will result in the sound finishing as soon as the sound duration is expired.

<cp> = Pitch change (16 bits) \ For each phase as described
<cl> = Left amp. change (8 bits) \ above, repeated up to 40
<cr> = Right amp. change (8 bits) / times (as specified by EP)
<pd> = Phase duration (16 bits) /

When an envelope definition is received, if an existing definition of that envelope exists then it is deleted. The new definition is then added to the list. If there is insufficient space to store it then an error code will be returned. If this happens then the old definition of that envelope will be lost.

4. Sound Production

To actually produce a sound an escape sequence must be sent which specifies the sound. The format of this is:

esc S <env> <p> <vl> <vr> <sty> <ch> <d> <f>

The meaning of each field is:

<env> - (8 bit) Envelope to use for this sound. An envelope number of 255 will produce a "beep" type sound which is of constant amplitude and pitch for the duration of the sound.

<p> - (16 bit) Starting pitch of sound in 1/512 semitones. Only exact quartertones will necessarily be musically correct. The others are generated by linear interpolation. Ignored for noise channel.

<vl> - (8 bit) Overall left amplitude. (0...255)
<vr> - (8 bit) Overall right amplitude. (0...255)

<sty> - (8 bit) Sound style byte. For the noise channel, this byte is put into the noise control register for the duration of the sound. For a tone channel the top four bits are put into the four sound control bits in the sound frequency register for that channel, they thus control filtering, distortion and ring modulation. Zero gives a pure tone or white noise.

<ch> - (8 bit) Source for this sound. 0, 1 or 2 for the appropriate tone channel and 3 for the noise channel.

<d> - (16 bit) Duration of this sound in ticks.

<f> - (8 bit) Flags byte.
b0...b1 - SYNC count for this sound. See later section on synchronisation.
b2...b6 - Not used, should be zero.
b7 - Set to force over-ride of any sound in queue for this channel. Clear to wait its turn.

When a sound is received it is added to the end of the appropriate queue (killing the queue first if bit-7 of the flags byte is set). If the queue is full then there are two courses of action which depend on the state of the EXOS variable WAIT_SND. If this is zero then the sound driver will just wait until there is space in the sound queue, testing the stop key to allow it to be interrupted. If this EXOS variable is non-zero then it will return an error code .SQFUL.

5. Processing of Sounds

5.1 Pitch and Amplitude Control

If the envelope specified in a sound definition is not defined then the appropriate channel will be silent for the duration of the sound. The same thing applies if the envelope definition vanishes during the course of processing the sound.

The production of a sound under control of an envelope requires various current values to be kept. There will be the current pitch value which is initialised to the pitch value given in the sound specification. There will also be left and right current amplitude values, which are initialised to zero at the start of the sound.

With these values initialised, processing of the sound can begin. Each phase of the envelope in turn is executed, each one lasting for the number of ticks specified in the envelope specification. At each tick, the current pitch and amplitude values are modified so that the change value in the envelope specification is spread linearly over the duration of the phase. The result at the end of each phase is that the signed change value has been added to the current value at the start of the phase, for each of the three parameters.

In the case of amplitude parameters the value is limited to the range 0 to 63. If an attempt is made by an envelope to make the amplitude go above 63 then the actual amplitude will just stick at 63. For pitch values there is no checking, the value will simply wrap around from 65535 to 0 and vice versa.

At each tick the current values of pitch and amplitude must be output to the sound chip itself. The details of how the register values are calculated are different for pitch and amplitude values.

For amplitude, the current left amplitude (6 bits) must be multiplied by the overall left amplitude specified in the sound definition (8 bits). The resulting 14 bit number is the required left amplitude. The top six bits of this value are be written to the appropriate DAVE register. The right amplitude is of course treated similarly.

The 16-bit pitch value must go through a logarithmic conversion process to produce in a counter value to go into the appropriate DAVE registers. The top byte of the pitch defines a quater-tone number from 0 to 255 (range 9-10 octaves). The counter value for this quarter-tone is found from a lookup table, with appropriate shifting depending on the octave. The top four bits of the lower byte of the pitch value are used to linearly interpolate between the selected counter value and the next one up. This gives a resolution of approximately 1/64 tone which is certainly adequate to produce smooth pitch changes.

5.2 Time control

While all the above processing of phases is going on, another counter is timing the length of the note itself. This is a 16 bit counter initialised to the sound duration specified in the sound definition, and decremented at each tick. If the end of the envelope is reached before this counter reaches zero then the sound will be silenced and remain silent until the counter does reach zero.

If the sound duration counter reaches zero before the envelope has finished then two things happen. Firstly if the release phase of the envelope has not yet been reached then control skips to the start of the release phase. Secondly a flag is set to allow this sound to be over-ridden by another sound which is waiting in the queue or appears in the queue at a later time.

5.3 Synchronisation

The flags byte in the sound definition has two fields defined, the OVER-RIDE bit and the SYNC count. If the over-ride bit is set then the relevant sound queue is flushed before putting the new sound in, so it will be ready to start immediately. If it is clear then the new sound is just added to the queue to wait its turn.

The SYNC count is a two bit count which is ignored when the sound is put in the queue. The sound driver maintains an internal SYNC COUNT which is normally zero. When a sound reaches the head of a queue and is thus ready to be played, the SYNC byte in the sound is examined in conjunction with the internal SYNC counter.

If the SYNC count in the sound is zero then the sound is just started in the normal way with no synchronisation. If the SYNC count in the sound is non-zero then the sound is held up and the internal SYNC counter is examined. If it is zero then the SYNC counter from the sound is copied into it. If it is non-zero then it is decremented by one and if it goes to zero then all held up sounds are released simultaneously.

The effect of all this is that if three sound are to be played simultaneously then they should be queued up on their appropriate channels each with a SYNC count of two (one less than the number of sounds). The sound driver will then ensure that they are started simultaneously even if one of them gets to the head of its queue slightly earlier. This facility can thus be used to iron out slight timing differences in multi voice tunes.

6. Other Control Functions

There are various other functions which the sound device provides which are listed here.

^Z - Flush all sound queues.

Esc Z <n> - Flush an individual sound queue.
n = 0, 1 or 2 for tone channel, 3 for noise channel.

^X - Flush all of envelope storage

^G - Cause a PING, see terminal bell below.

7. Interaction With Other Devices

7.1 Other Devices Accessing Sound Registers

Ideally the sound device would have exclusive use of all the sound registers. Unfortunately the cassette device has to fiddle with some of them for output and timing purposes. The sound driver sets up all sixteen sound registers on each interrupt so it will recover very quickly if its registers are corrupted.

7.2 Keyboard Click

The keyboard device has to click when a key is pressed. This event must be triggered from the keyboard device's interrupt routine and so cannot be done with an EXOS call. It is achieved by the keyboard device calling a globally defined routine KEYCLICK in the sound driver which uses tone channel zero to produce a click without interfering with any other sound which may be on this channel. The key click sound takes about 1/100 second and the routine does not return until it has finished.

5.3 Terminal Bell

When the screen driver gets an ASCII code 7 (BELL) it is supposed to make an appropriate sound. It does this by calling a globally defined routine called BEEP in the sound driver. This uses tone channel 2 to make a bell sound. If there is already some sound on this channel then the routine does nothing. If there is no sound on this tone channel then a ping sound will be started and the routine will return. If another call to BEEP is made before the first ping has finished then it will hang up until the first one finishes, and then start the second one and return. .cp 3

A ping can also be triggered by sending an ASCII BELL character to the sound driver. This will have just the same effect as if the video driver calls the routine BEEP.

PROGRAMMING THE DAVE CHIP

****

The DPC Sound Chip performs the following functions:-

1. Multi-function '3 tones + noise' stereo sound generator.
2. Memory paging.
3. Address decoding for on-board ram, rom and cartridge.
4. Interrupt system including 1Hz and programmable frequency timer interrupts and two external inputs.
5. Reset circuit compatible with Z80 and dynamic ram.
6. I/O strobe signals for use with external octal latches and tri-state buffers.
7. 1MHz system clock.
8. Z80 wait state generator.

DPC Sound chip has 22 internal registers, 17 of which are write-only. 16 of these registers are associated with the sound generation, four R/W registers are for memory management, and one R/W register is used for interrupt control. The last write-only register is used for setting the overall system configuration. Internal decoding is provided for a further 3 I/O registers, read and write strobes being brought out for use with external latches and tri-state buffers on the data bus. Reset clears all 22 internal registers.

The 3 tone generators produce square waves with frequency programmable from 30Hz to 125KHz which can be modified in various ways:-

a. Distortion can be introduced by using the output frequency to sample H.F. clocked polynomial counters. PN counters which can be selected are 4,5 or 7 bit. The 7 bit PN can also be exchanged for a variable length 17/15/11/9 bit PN counter.
b. A simple high pass filter is provided on each channel, clocked by the output of a different channel.
c. A ring modulator effect is provided on each channel, with the output of a different channel for it's other input.

The noise channel is normally a 17 bit PN counter clocked from 31KHz, generating a pseudo white noise. The input to this counter can be changed to clock off any of the 3 tone channels, and the PN counter can be reduced in length to 15, 11 or 9 bits. This counter can also be exchanged for the 7 bit PN counter. The resulting noise is then passed through high pass and low pass filters and a ring modulator, each controlled by the output of a different tone channel.

The 3 tone generator outputs and the noise generator output are routed to 2 amplitude control circuits (left and right). Each amplitude control consists of four 6 bit write-only registers (one for each sound) which are multiplexed onto an external 6 bit D/A resistor network. In it's own time slot each channel outputs the value in it's amplitude register if tone is high, else zero.

Either or both of the sound output channels may be turned into 6 bit D/A outputs, when they will constantly output the values in tone channel 0 amplitude registers. This is controlled by 2 bits in the write-only sound configuration register. Three further bits may be used to synchronise the tone generators by holding them at a preset count until sync bit goes low.

Memory management consits of 4 read/write registers which may be output onto A14-A21 pins by selecting the required register with A14', A15'. This provides 256 * 16K pages. These outputs may be tri-stated with BREQ.

Four latched interrupts are provided, a 1Hz interrupt for time clock applications, an interrupt switchable between 50Hz, 1KHz, or the outputs of tone generators 0 or 1, and two external negative edge triggered interrupts. Each interrupt latch has it's own enable and reset controlled by a 8 bit write-only register. An attempt to read this register will return the state of the four interrupt latches and two interrupt input pins, and also two flip -flops toggling off the timer interrupts. The setting of any interrupt latch will bring IRQ low (open drain). 50Hz/1KHz/tone generator interrupt selection is made by 2 bits in the sound configuration register.

Select signals are generated for rom, cartridge, video ram and video I/O. A 1MHz clock output is also provided.

A Z80 reset is provided on RSTO, either on switch on by an external RC network on CAP, or a low going signal on RSTI. The latter generates a 1mS reset pulse synchronised to the falling edge of M1 to prevent loss of data stored in dynamic ram. The RSTO output requires an external 74ALS04 inverter to drive the system reset line at the correct speed and inversion.

A write-only system configuration register is used to set the system for 16/64K on board ram, 8/12MHz input clock, and wait states. The wait state generator can be programmed to give no wait states, waits on opcode fetch only, or waits on all memory accesses. Note that no wait is ever generated for access to video ram, as this would conflict with Z80 clock stretch.

REGISTER DESCRIPTIONS

R0 W £A0

b7-b0 Low byte of number to be loaded into 12 bit down counter to set period of tone channel 0.

R1 W £A1

b3-b0 High nybble of above, f out = 125,000/(n+1) Hz.

b5,b4
00 = Pure tone
01 = Enable 4 bit polynomial counter distortion
10 = " 5 bit " " "
11 = " 7 bit " " "

b6 1 = Enable high pass filter using tone channel 1 as clock.

b7 1 = Enable ring modulator with tone channel 2

R2 W £A2

As R0 but for tone channel 1.

R3 W £A3

As R1 but for tone channel 1 except:-

H.P.F. uses tone channel 2
R.M. uses noise channel

R4 W £A4

As R0 but for tone channel 2.

R5 W £A5

As R1 but for tone channel 2 except:-

H.P.F. uses noise channel
R.M. uses tone channel 0

R6 W £A6
----

b1,b0 Select noise clock frequency:-

```
00 = 31.25KHz
01 = tone channel 0
10 =  "     "    1
11 =  "     "    2
```

b3,b2 Select polynomial counter length:-

```
00 = 17 bit
01 = 15 bit
10 = 11 bit
11 =  9 bit
```

b4 1 = Swop 17 bit and 7 bit polynomial counters

b5 1 = Enable low pass filter on noise using tone channel 2 as clock.

b6 1 = Enable high pass filter on noise using tone channel 0 as clock.

b7 1 = Enable ring modulator with tone channel 1

R7 W £A7
----

b0 Sync for tone channel 0. (1 = hold at preset, 0 = run)

b1 Sync for tone channel 1.

b2 Sync for tone channel 2.

b3 1 = Turn L.H. audio output into D/A, outputting value in R8.

b4 1 = Turn R.H. audio output into D/A, outputting value in R12

b6-b5 Select interrupt rate:-

```
00 = 1KHz
01 = 50Hz
10 = Tone generator 0,     f=250,000/(n+1)
11 = Tone generator 1
```

b7 Undefined

R8 W £A8
----

b5-b0	Tone channel 0 L.H. amplitude Also value output to L.H. D/A if R7 b3 = 1
b7,b6	Undefined.

R9 W £A9
----

b5-b0	Tone channel 1 L.H. amplitude
b7,b6	Undefined.

R10 W £AA
----

b5-b0	Tone channel 2 L.H. amplitude
b7,b6	Undefined.

R11 W £AB
----

b5-b0	Noise channel L.H. amplitude
b7,b6	Undefined.

R12 W £AC
----

b5-b0	Tone channel 0 R.H. amplitude Also value output to R.H. D/A if R7 b4 = 1
b7,b6	Undefined.

R13 W £AD
----

b5-b0	Tone channel 1 R.H. amplitude
b7,b6	Undefined.

R14 W £AE
----

b5-b0	Tone channel 2 R.H. amplitude
b7,b6	Undefined.

R15 W £AF
----

b5-b0	Noise channel R.H. amplitude
b7,b6	Undefined.

R16 R/W £B0
----
b7-b0 Page register output to A21-A14 if A15',A14' = 00

R17 R/W £B1
----
b7-b0 Page register output to A21-A14 if A15',A14' = 01

R18 R/W £B2
----
b7-b0 Page register output to A21-A14 if A15',A14' = 10

R19 R/W £B3
----
b7-b0 Page register output to A21-A14 if A15',A14' = 11

R20 W £B4
----

b0	1 = Enable 1KHz/50Hz/TG interrupt.
b1	1 = Reset 1KHz/50Hz/TG interrupt latch.
b2	1 = Enable 1Hz interrupt.
b3	1 = Reset 1Hz interrupt latch.
b4	1 = Enable INT1.
b5	1 = Reset INT1 latch.
b6	1 = Enable INT2.
b7	1 = Reset INT2 latch.

R20 R £B4
----

b0	1 = 1KHz/50Hz/TG divider. (f int/2 square wave).
b1	1 = 1KHz/50HZ/TG latch set.
b2	1 Hz divider. (0.5 Hz square wave).
b3	1 = IHz latch set.
b4	INT1 input pin.
b5	1 = INT1 latch set.
b6	INT2 input pin.
b7	1 = INT2 latch set.

R21 W £B5

Active low strobe on WR0.

R21 R £B5

Active low strobe on RD0.

R22 W £B6

Active low strobe on WR1.

R22 R £B6

Active low strobe on RD1.

R23 W £B7

Active low strobe on WR2.

R23 R £B7

Active low strobe on RD2.

R31 W £BF

```
b0     On board RAM, 0=64k,1=16k.

b1     Input clock frequency, 0=8MHz, 1=12MHz.

b3,b2  00 = Wait on all memory access except video ram.
       01 = Wait on M1 only, except video ram.
       10 = No waits.
       11 = No waits.
```

179

SELECT OUTPUTS

VIO Low for I/O access £80 to £8F. Gated with IORQ,RD,WR in video chip.

ROM Low for memory access on pages 0-3. (0-£FFFF) Gated externally with RD.

CART Low for memory access on pages 4-7. (£10000-£1FFFF) Gated externally with RD,WR

VRAM Low for any memory access on pages £FC-£FF. (£3F0000-£3FFFFF) IF R31 b0 = 0

Low for any memory access other than rom or cartridge, (£20000-£3FFFFF) IF R31 b0 = 1

Gated with MREQ,RD,WR in video chip.

END OF DOCUMENT