MD.SECTR Vector 12A

Read a sector header on a microdrive

Call parameters		Return parameters	
D1		D1	
D2		D2	
D7		D7	sector number
A0		A0	
A1	pointer to buffer start	A1	pointer to buffer end
A2		A2	undefined
A3	$18020	A3	$18020

Error returns:

normal = bad medium
return+2 = bad sector header
return+4 = OK

Description:

This routine reads a sector header on a microdrive. All registers except A3 and A6 are volatile. A3 must be set to point to the microdrive control register at entry and the interrupts must be disabled. D0 is not set on return, but there are three possible returns. The normal return implies that the medium was bad. The normal return plus 2 means that there was a bad sector header and the normal return plus 4 indicates that the sector header was successfully read.

The vector to this routine points to $4000 bytes before the actual code. The following code may therefore be used to invoke it:

```
MOVE.W      aa.aaaa,An
JSR         $4000(An)
RET1.W      return at this point means bad medium
RET2.W      return here means bad sector header
RET3.W      return here means successful read
```


9 Device Drivers

9.1 Introduction

This chapter deals with the ways in which physical devices are linked into the QL operating system. As will have been appreciated from earlier chapters, the QL has a very powerful redirectable I/O system. In order for this I/O system to work equally well with a large variety of hardware devices, it is necessary to have a well defined interface mechanism. This interface mechanism is provided by *device drivers*.

The device driver sits in the middle of two processes. On the one hand there is the QL IOSS (I/O subsystem). This expects the device driver to be able to open and close channels and input or output data for it. On the other hand, the device driver must be able to control particular pieces of hardware. Control is normally carried out following an interrupt from a device, or on the QL's 50/60Hz interrupt or alternatively on being invoked by the scheduler.

The device driver can therefore be considered as the interface between two separate layers. The *access layer* communicates with the IOSS. Several entry points to the device driver from the IOSS are provided. There is one to open a channel, one to close a channel, and a bidirectional one to allow data transfer between the driver and IOSS. The other layer is called the *physical* layer. The essential functions of this layer are control of a hardware device and data communication with that device.

It is inherent that the *physical layer* of any device driver will be of a highly specialised nature. This is because most types of hardware devices will be very dissimilar. However, the *access layer* interfacing procedure has been very clearly defined and should be adhered to, irrespective of the hardware being driven.

A device driver must maintain some space in RAM where it can store certain types of information. Each device driver is allocated a *Device driver definition block*. This is either in the system variables area for inbuilt device drivers, or in some other part of memory (with restrictions on where) for additional drivers. This

block of RAM is available as the *static* working area of the device driver, and contains information like the number of channels open to the device driver, the addresses of any open channels (or a pointer to a linked list of channels if the number could be infinite), the current status of the I/O hardware etc. This area is *owned* by the device driver, and therefore by the operating system of which it is a part.

Now, it is possible for several channels to be open to a device driver at once. The device driver must therefore be re-entrant. Clearly, if the device driver tried to cope with the current status of each channel by itself, it would soon get into a mess. For this reason, each channel is allocated a *channel definition block*. This block is created whenever a channel is opened by IO.OPEN (trap#2 D0=1), and is then owned by the Job which created it.

The basic concepts discussed above are now dealt with in much greater depth in the following sections.

9.2 Device driver definition blocks

Device drivers are normally linked into the operating system when the machine is powered up. A series of manager traps (trap#1 with D0=1A to 23) are available to do this. It is not necessary to link in device drivers at power up – they can be added later if required. Linked lists are maintained by the manager for the various operations. Linked lists have a standard form and are discussed in some detail in section 5.4.5.

There are five linked lists in all. Three of these are for the physical layer of device drivers as invoked by:

1. external interrupts
2. 50/60 Hz interrupt
3. scheduler loop

As described in section 5.4.5, a linked list is produced by reserving two long words. The first forms the link in the list. The second should hold the address of the physical device driver code. These two long words normally come at the start of a memory block. The rest of the block is then available for use by the device driver. Some uses might be the storage of flags, pointers, buffers etc.

The operating system passes the physical device drivers several useful pieces of information. These are contained in the 68008 registers as follows:

D3	number of 50/60 Hz interrupts since last scheduler call (scheduler loop only).
A3	pointer to device driver definition block
A6	pointer to system variables
A7	supervisor stack (64 bytes can be used)

Both A6 and A7 must be preserved by the device driver.

For access layer calls, A3, A6 and A7 have the same meaning, but the other registers are different (see 9.4).

A3 points to the base address of the definition block. This refers to a standard device driver definition (see below). It is however possible to have a different, non-standard definition (for example some of the entry addresses and link pointers could be left out). If a non-standard definition is used, A3 will not have such a convenient value.

$00(A3)	link to next external interrupt routine
$04(A3)	address of external interrupt routine
$08(A3)	link to next 50/60Hz interrupt routine
$0C(A3)	address of 50/60Hz interrupt routine
$10(A3)	link to next scheduler loop routine
$14(A3)	address of scheduler loop routine
$18(A3)	link to access layer of next device driver
$1C(A3)	address of input/output routine
$20(A3)	address of channel open routine
$24(A3)	address of channel close routine
$28(A3)	physical device driver working space

9.3 Access layer

To initialise the access layer, a block of memory must be linked into the IOSS using MT.LIOD (trap #1 with D0=$20). This block of memory should be in the resident procedure or common heap area. There must be four long words at the start of the block. The manager uses the first to form the linked list. The second, third and fourth long words of an access layer driver are the entry addresses for input/output, opening and closing a channel respectively. The rest of the block can be used for containing pointers, buffers etc.

The format for the linked block is:

long word 1 ink pointer to next linked routine
long word 2 entry address for input/output
long word 3 entry address to open a channel
long word 4 entry address to close a channel

Note that all access layer calls are made in supervisor mode.

9.3.1 Opening a channel

When a call is made to open a channel, A0 contains a pointer to the device name and D3 contains a key. The assumed base of the device driver definition block is held in A3. A6 points to the system variables area and A7 is the supervisor stack pointer. Registers D0 to D7 and A1 to A6 may be treated as volatile.

The sequence leading to a successful channel open operation is:

Decode device name.
Allocate channel definition block and buffers in common heap.
Initialise channel definition block (except first 6 long words).
Return address of channel definition block in A0.

All channels are bidirectional. It is the device driver's responsibility to trap illegal operations. Four common error returns in D0 can occur when an attempt is made to open a channel. **ERR.NF** means that the device name was not

recognised. **ERR.IU** means that the required IO device is being used by another Job and cannot be shared. **ERR.BN** means that the device name was recognised but that some of the additional information was incorrect in value or format: **ERR.OM** means that everything was correct, but the channel definition block or buffer could not be set up due to lack of memory.

9.3.2 Inputing and outputing to a channel

The input/output call is first made during an I/O trap. If the trap has specified a *wait until complete* return, the input/output call is made once on every scheduler loop. This continues until the operation is complete, or until it times out.

A0 is set to point to the channel definition block before making an I/O call. D0 defines the required operation (this is a byte key and the IOSS clears the upper three bytes). Additional information is passed in D1, D2, A1 and A2. Register D1 is set to zero by the IOSS for all string operations (IO.FSTRG, IO.FLINE, IO.SSTRG, FS.HEADS, FS.HEADR, FS.LOAD and FS.SAVE). If an operation is incomplete, the values in D1 and A1 must be set on exit so that they can be used again directly for the re-try. To assist in this process, D3 is set to zero on the first entry, but is modified to −1 on the second and subsequent entries. D0, D2 and A2 keep their original values throughout the process.

D0 always defines the operation, A3 points to the assumed base of the device driver definition block, A6 points to the system variables area and A7 is the supervisor stack pointer.

Within the device driver, registers D2 to D7 and A2 to A6 can be modified at will.

9.3.3 Closing a channel

A close channel operation checks that everything is tidy and then releases the space which was allocated to the channel definition block and buffer. If there are still some bytes left to be transmitted, it may not be possible to release the space immediately. In such cases, the block will be deleted by a subsequent scheduler call when the data has been transmitted.

The close routine is passed the address of the channel definition block in A0. The assumed base of the device driver definition block is pointed to by A3. A6 points to the system variables area and A7 is the supervisor stack pointer. Registers D1 to D7 and A0 to A6 can be changed by the code as required. D0 should be set to the error return. In general, this should be zero (ie. no error) since a lot of the system assumes that a close never fails.

9.4 Physical layer

As was mentioned in section 9.1, the physical layer has the function of interfacing between the device driver and external hardware. This interface may be carried out on any one of the following events:

1. External interrupts

External interrupts are generated by particular hardware devices. If a device driver expects external interrupts from the device which it is serving, it should first of all check the hardware status to see if the interrupt was due for it. If the interrupt was not from the correct device, a return should be made via an RTS. If the interrupt was intended for the device driver, the hardware should be serviced as required and the source of the interrupt should be cleared. A return should then be made to the main routine via an RTS instruction. Note that atomic operations (like access layer calls to the same driver) can be interrupted by this interrupt.

2. 50/60 Hz interrupts

This interrupt should only be used for critical timing operations, or opertions which should not be held up for long periods. An example of a critical timing operation would be the incrementing of a register to keep track of time, or the generation of sound envelopes. If the external hardware cannot interrupt the system when it needs to be serviced, the 50/60 Hz interrupt could be used to poll the relevant device to see if servicing is required. Note that atomic operations (like access layer calls to the same driver) can be interrupted by this interrupt.

3. Scheduler loop

Calls from the scheduler loop will come round at irregular intervals (depending upon the workload on the system). Anything which must be processed immediately should not therefore use this loop. Atomic operations will not be interrupted by the scheduler, so operations such as allocating or releasing memory can safely be performed by routines running off the scheduler.

General notes on the physical layer

The physical layer allows data to be transferred into or out of queues or to execute control functions asynchronously with a Job requiring I/O. All physical layer routines are called in supervisor mode, so it is perfectly in order for them to disable interrupts. Any errors should normally be processed by setting a flag which will be dealt with by the access layer. However, it is possible to use the utility routines UT.ERR and UT.MTEXT with A0=0. An attempt will then be made to write a message to channel 0, or, failing that, to channel 1. The message will only appear on one of these channels if a channel is not waiting for input. If both channels are waiting for input, no message will appear. The queue handling routines have been designed so that data can be transferred to/from a queue asynchronously.

9.5 Decoding the device name

The IOSS utility IO.NAME (vector $122) is a useful aid to help with the decoding of device names. It will perform two functions by checking the device name and then evaluating any optional parameters.

The full device name contains four components:

1. Name	ASCII characters, normally letters and case independent.
2. Separator	ASCII character, if it's a letter then the case is ignored.
3. Number	Decimal number in the range 0 to $2^{15} - 1$.
4. Code	One of a list of ASCII characters.

A0 should contain a pointer to the actual device name when IO.NAME is called. A3 should contain a pointer to a block of

memory which is large enough to store the parameter values. If the routine is successful, the block will be filled, either with the given, or with the default parameters.

IO.NAME has three possible returns:

return	D0 (+ SR)
standard	ERR.NF name not recognised
standard+2	ERR.BN name recognised, bad parameter
standard+4	0 successful return

The device name description then commences 6 bytes after the call.

9.5.1 Device name description

Description is of the following form:

Number of character in name, characters in name
Number of parameters
For each parameter one of:

Space + separator, default value (numeric)
Negative number, default value (number, no separator)
Number of codes, list of ASCII codes.

All items are defined as **words** and all letters must be upper case. Since the ASCII codes for characters actually fit into bytes, any odd byte at the end is filled with a zero. For example:

```
DC.W 3,'ABC'
```

would be put into memory as:

word1 $00 03
word2 $41 42
word3 $43 00

For each numeric parameter value in the description, the utility will return either the given value or the default. For each code list in the description the utility will return the position of the code in the list or zero.

Examples

The CON description is:

DC.W 3,'CON'	console
DC.W 5	five parameters
DC.W '_',448,' X',200	window size
DC.W ' A',32,' X',16	window position
DC.W '_',128	keyboard buffer length

Device name	Returned parameters
CON	448,200,32,16,128
CON_256	**256**,200,32,16,128
con__60	448,200,32,16,**60**
cona0x12	448,200,**0,12**,128
con_256x64a64x128_20	**256,64,64,128,20**

The SER description is:

DC.W 3,'SER'	RS232 serial device
DC.W 3	three parameters
DC.W −1,1	port number (default 1)
DC.W 4,'EOMS'	parity even, odd, mark, space
DC.W 2,'IH'	ignore/use handshaking

Device name	Returned parameters
SER	1,0,0
sere	1,**1**,0
ser2mi	**2,3,1**

9.6 Memory Allocation

It is essential that the access layer drivers are fully re-entrant. This means that all channel specific information must be stored in the channel definition block and not in the device driver definition block. Channel definition blocks require 6 long words to be left at the beginning for use by the IOSS. The remainder of the space in the channel definition block can be used by the device driver.

Physical layer drivers **must never** allocate or release machine resources unless this is done on a scheduler call.

Space should always be allocated in the common heap area. Routine MM.ALCHP can be used to allocate space. This space can be released again by routine MM.RECHP which releases common heap memory.

9.7 I/O Queues

There are five resident queue handling routines. These control the entry of data into and extraction of data from a queue, as well as checking the contents of the queue, or end of file condition. IO.QSET sets up the queue pointers. IO.QIN puts a byte in a queue. IO.QOUT takes one out. IO.QEOF puts an EOF flag in and IO.TEST checks if the queue has anything in it, returns the free space in the queue and the value of the next byte to be taken out without actually removing the byte from the queue.

Queues are defined by a block of four long words at the start. The MSB of the first long word is used by the queue routines to signify the end of a file. IO.QSET clears the entire long word, the remainder (ie. without most significant bit) can be used by device drivers for linking in queues etc.

The usable length of a queue is one byte less than the actual length of a queue. The minimum space occupied by a queue and it's header is therefore 18 bytes.

9.8 Simple serial I/O

Simple serial I/O can be handled by IO.SERQ. For this routine, the 7th and 8th long words in the channel definition block should be set to point to the queues for input and output respectively. If either input or output is prohibited then the corresponding pointer should be set to zero. For undefined actions error ERR.BP will be returned. See the description of IO.SERQ in chapter 8 for more details.

In cases where the operations of byte input and output are not quite so simple, the routine IO.SERIO may be called. The call instruction should be followed by three long words, these being the entry addresses for:

1. Testing for a pending input (next byte in D1)
2. Fetch a byte (byte in D1)
3. Send a byte (byte in D1)

9.9 Directory device drivers

Directory device drivers are more powerful in their application than pure device drivers. The ordinary device drivers are only responsible for communication with one device. Directory device drivers are responsible for communicating with particular files on a device, such as a Winchester hard disc. The IOSS has an extended range of operations for directory device drivers.

9.9.1 Directory driver definition block

Definition blocks for directory device drivers are similar to those for device drivers. However, there are three parts; the directory driver linkage block (one per directory driver), the physical definition block (one per drive) and the channel definition block. The IOSS allocates the latter two.

The linkage block

The standard form is an extended version of that used for device drivers. The access layer linkage is at least 10 long words in length and has the standard format of:

$00(A3)	link to next external interrupt routine
$04(A3)	address of external interrupt routine
$08(A3)	link to next 50/60 Hz interrupt routine
$0C(A3)	address of 50/60 Hz interrupt routine
$10(A3)	link to next scheduler loop routine
$14(A3)	address of scheduler loop routine
$18(A3)	link to access layer of next directory driver
$1C(A3)	address of input/output routine
$20(A3)	address of channel open routine
$24(A3)	address of channel close routine
$28(A3)	address of entry for forced slaving
$2C(A3)	reserved
$30(A3)	reserved
$34(A3)	address of entry to format medium
$38(A3)	length of physical definition block
$3C(A3)	word – contains length of drive name
$3E(A3)	characters of drive name (eg. mdv)

9.9.2 Directory driver access layer

As for device drivers, the access layer has to open, close and input or output to files. Close and input/output are the same for directory device drivers and device drivers. Refer to sections 9.3.2 and 9.3.3 for more details on this. Opening channels on a directory device driver is however a bit more complex.

Opening a channel

Before calling the channel open routine, the channel definition block and the physical definition block for the appropriate drive must have been allocated. At this initial stage, the IOSS must have partially filled both of these blocks. On entry to the open routine, A0 will point to the channel definition block and A1 will point to the physical definition block. A3 will point to the assumed base of the linkage block. The physical definition block may or may not have just been allocated. The registers D0 to D7 and A1 to A5 can be changed. Files are comprised of blocks of 512 bytes each. An actual byte position is therefore 512*block number + byte number.

Channel definition block

Channel definition header for all channels

$00	CH.LEN	long	length of definition block
$04	CH.DRIVR	long	address of driver
$08	CH.OWNER	long	owner Job
$0C	CH.RFLAG	long	address to be set when space released
$10	CH.TAG	word	channel tag
$12	CH.STAT	byte	status: 0 = OK −1 = A1 absolute $80 = A1 relative to A6 negative = waiting
$13	CH.ACTN	byte	stored action for waiting Job
$14	CH.JOBWT	long	ID of Job waiting on IO

File system channel definition block

$18	FS.NEXT*	long	link to next file system channel
$1C	FS.ACCES*	byte	access mode (D3 on open call)
$1D	FS.DRIVE*	byte	drive ID
$1E	FS.FILNR+	word	number of file on drive
$20	FS.NBLOK	word	block number containing next byte
$22	FS.NBYTE+	word	next byte from block
$24	FS.EBLOK+	word	block number containing byte after EOF
$26	FS.EBYTE+	word	byte after EOF
$28	FS.CBLOK	long	pointer to slave block table for current slave block which may hold current/next byte
$2C	FS.FNAME*	2+36 bytes	filename
$58	FS.SPARE	72 bytes	spare

Physical definition block

$10	FS.DRIVR*	long	pointer to access layer link for driver
$14	FS.DRIVN*	byte	drive number
$15		byte	reserved
$16	FS.MNAME	word+10 bytes	medium name
$22	FS.FILES	byte	number of files open

Length is determined by the access layer linkage.

Key

* Initially the blocks are full of zeros apart from those marked with an * which are filled in by the IOSS

+ The open routine is not called if a shared file is open to more than one channel. In this case, FS.NBYTE is set to $40 and the fixed information is copied from the first channel open to the file.

The drive ID is an index to a table of physical definition blocks starting at $100 on from the base of the system variables.

Deletion of files is part of the channel open operation. The two processes are distinguished by the access key being negative for deletion. Definition blocks are allocated in the usual manner for opening a file, but the IOSS releases the channel definition block on return. The channel definition block is also released by the IOSS if an error code is returned in D0.

9.10 Slave blocks

All interaction with storage devices is carried out through buffering in the QL's memory. This usually goes on as a background process. If a microdrive file is accessed, several sectors will be read into memory. Rewriting the sectors to the microdrive does not necessarily occur immediately that a save is performed. Indeed, data can remain in memory for sometime waiting to be dumped to a store. As the space used by transient programs, resident procedures and the heaps uses up memory, so the slave block area decreases in size.

Space in the slave blocks is defined by a slave block table. This table is of a variable size, and depends largely upon the amount of spare memory. There are three pointers to the slave block table.

$54(A6)	SV.BTPNT	long	pointer to most recently allocated entry in the table
$58(A6)	SV.BTBAS	long	pointer to the base of the table
$5C(A6)	SV.BTTOP	long	pointer to the top of the table

Each table entry is 8 bytes long. Only the first byte is used by the operating system – this is the status byte. The other bytes are not used by the operating system.

$00	BT.STAT	byte	status byte
		00	unavailable to filing system
		01	empty block
		X3	block is a true representation of file
		X7	block is updated (awaiting write)
		X9	block is awaiting read
		XB	block is awaiting verify
		X	is the drive ID for this file
$01	BT.PRIOR	byte	available for fancy slaving algorithms
$02	BT.SECTR	word	physical sector on medium
$04	BT.FILNR	word	file number
$06	BT.BLOCK	word	block number of contents of this slave block

Since slave block allocation is simply a form of allocating memory, it should only be done by access layer calls or scheduler loop calls.

It will be necessary to search for a slave block to use as a buffer for a file block. Only blocks which are empty or true representations should be used. When a new block is allocated to a file, SV.BTPNT should be updated.

Never assume that blocks which were allocated on one call are still allocated on the next call because they might not be! FS.CBLOK is purely an advisory variable. **Always** check the slave block table entry before proceding.

Sometimes the memory manager will require the space occupied by an updated block or a block awaiting read or verify. If this occurs then the forced slaving routine will be called to convert it into a true copy. The routine doesn't check that the block is a true copy, but initiates the process which will check it. On entry to the forced slaving routine A1 points to the offending slave block. A2 points to the physical definition and A3 points to the assumed base of the linkage block (driver definition). Registers D0 to D3 and A0 to A4 can be altered if desired.

9.11 A parallel printer driver example

9.11.1 Introduction

This section illustrates many of the points covered in earlier parts of the chapter. A complete example of a device driver to implement a standard Centronics printer interface is discussed. The complete circuit diagram for this circuit is in Appendix U.

This device driver has been designed to fit into a ROM on a peripheral card. If such a card (with a ROM) is plugged into the QL, the message 'Simple Centronics Interface' will be printed on the screen at power up. From then on, a new channel called 'PAR' will exist. To list a microdrive file on a printer, it would simply be a matter of typing

```
copy mdv1_textfile to par.
```

Since this is only a simple example of a device driver implementation, no buffering of the printed data is provided. This means that the driver will keep on trying to output a byte, and returning with an 'ERR.NC' if the byte was not sent (due to the printer being busy). A more sophisticated driver would operate a buffer and output on interrupts generated from the interface circuitry.

9.11.2 The device driver program

```
*
*  Centronics Parallel Interface Driver (non interrupt)
*
ERR.NC    EQU    -1
ERR.BP    EQU    -15
MT.ALCHP  EQU    $18
MT.LIOD   EQU    $20
IO.SSTRG  EQU    $7
MM.ALCHP  EQU    $C0
MM.RECHP  EQU    $C2
IO.SERIO  EQU    $EA
IO.NAME   EQU    $122
*
DATA      EQU    $0800    address of data latch
STROBE    EQU    $1000    address of data strobe
BUSY      EQU    $1800    address of BUSY buffer
*
BASE
          DC.L   $4AFB0001
          DC.W   0
          DC.W   INIT-BASE
          DC.B   0,28,'Simple Centronics Interface',$A
*
INIT
          MOVEM.L A0/A3,-(A7)   A0 & A3 used by init code
          MOVEQ   #MT.ALCHP,D0  linkage block heap allocation
          MOVEQ   #$36,D1       $28 + 3 long words + a word
          MOVEQ   #0,D2         owned by Job zero
          TRAP    #1
          TST.L   D0            check if OK
          BNE.S   INIT_EXIT
*
          LEA     $1C(A0),A3    start filling linkage block
          LEA     IO(PC),A2     input / output
          MOVE.L  A2,(A3)+      ... at $1C
          LEA     OPEN(PC),A2   open
          MOVE.L  A2,(A3)+      ... at $20
          LEA     CLOSE(PC),A2  close
          MOVE.L  A2,(A3)+      ... at $24
          LEA     ERR_BP(PC),A2 test pending input + get byte
          MOVE.L  A2,(A3)+      ... at $28
```



```
          MOVE.L  A2,(A3)+      ... and $2C
          LEA     OUT_BYTE(PC),A2 send byte
          MOVE.L  A2,(A3)+      ... at $30
          MOVE.W  #$4E75,(A3)+  RTS at $34
*
          LEA     $18(A0),A0    link into
          MOVEQ   #MT.LIOD,D0   IO driver list
          TRAP    #1
INIT_EXIT
          MOVEM.L (A7)+,A0/A3
          RTS
*
*
OPEN
          MOVE.W  IO.NAME,A4    decode device name
          JSR     (A4)
          BRA.S   EXIT          bad
          BRA.S   EXIT          bad
          BRA.S   ALCHP         name was 'PAR'
          DC.W    3,'PAR'       definition    name is PAR
          DC.W    0                           no parameters
ALCHP
          MOVEQ   #$18,D1       reserve the minimum channel
          MOVE.W  MM.ALCHP,A4
          JMP     (A4)
CLOSE
          MOVE.W  MM.RECHP,A4   remove channel
          JMP     (A4)
IO
          CMP.B   #IO.SSTRG,D0  trap file operations (e.g. set
          BHI.S   ERR_BP        ... header), because they make
*                               ... no sense on a printer
          PEA     $28(A3)       pretend we have a call at
          MOVE.W  IO.SERIO,A4   ... $24(A3) to SERIO
          JMP     (A4)
*
*
OUT_BYTE
          LEA     BASE(PC),A3   get peripheral board base addr
          TST.B   BUSY(A3)      is printer busy?
          BMI.S   ERR_NC        sign bit is busy
          MOVE.B  D1,DATA(A3)   send byte
          SF      STROBE(A3)    and strobe (>500ns)
          MOVEQ   #0,D0
EXIT
          RTS
ERR_NC
          MOVEQ   #ERR.NC,D0    byte has not been sent
          RTS
ERR_BP
          MOVEQ   #ERR.BP,D0    cannot fetch byte and
          RTS                   ... cannot do file operation
```

9.11.3 Program operation

In order to understand the device driver operation, it is important to realise how the hardware appears in the QL memory map. The 8-bit printer output port is located at address $0800 from the base address of the peripheral board. The data strobe output is at address $1000, and the BUSY input from the printer is at address $1800 from the base of the peripheral board.

For those who are not familiar with the peripheral board address decoding mechanism (Appendix F), the following explanation is in order. Up to 16 peripheral cards can be plugged into a QL. Each of these is assigned 16K bytes of memory at the top of the QL's memory map. The particular base address of any card is determined by its position in the motherboard. All peripheral addresses are therefore specified relative to the card base address. When the QL powers up, a check is made for the identification code $4AFB0001 at the start of each peripheral card address. If this code is found, a ROM is assumed to be present containing device drivers, functions and/or procedures.

In order to indicate to QDOS that this is a ROM containing a device driver, special recognition data must be present at the start (see Appendix C). The data from 'BASE' onwards will fit into the first few bytes. The first long word identifies that this is a ROM and not just 'empty memory space'. The word following this indicates that there are no procedures or functions in this ROM. The next word points to the initialisation routine in the ROM, which will be run once at power up. Finally, the length of the device driver name and characters making up the name are provided.

The initialisation routine 'INIT' allocates some space on the heap for the linkage block, fills in the relevant addresses and data, then links the block into the IO driver list. Note that A0 and A3 are saved at entry to INIT, and are restored at exit, since they are both modified by the routine.

As with all standard device drivers, the three standard routines are provided. These are open, close and input/output.

OPEN

Opening a channel consists of two major operations. First of all, the channel name has to be decoded. In this case, IO.NAME has been used to simplify this. Then, once the name has been recognised, $18 bytes of space are allocated in the common heap area. It will be necessary to allocate more than this in certain applications, but only the minimum is used here. The channel has now been opened.

CLOSE

Closing a channel is very easy. The operation consists of removing the allocated space in the common heap. MM.ALCHP is used to release the common heap.

Input/Output

This routine uses the addresses at $28(A3) onwards. These point to routines for checking pending input, fetching a byte and sending a byte respectively. The general string handling routine IO.SERIO is used to handle IO. Most IO operations are valid, except IO.SSTRG (this is concerned with file operations like setting a header, which isn't relevant to a printer).

The pending input and fetch byte routines return 'Bad parameter' error. Only the routine to send a byte to the printer (OUT_BYTE) is valid.

OUT_BYTE first of all checks to see whether the printer is busy. If it is busy, an ERR.NC is returned. If the printer is not busy, a byte is written to the data latch. This is then sent to the printer by operating the strobe line for >500ns. After this sequence of operations, the data has been transferred to the printer.