QL· Technical Guide

by Tony Tebby and David Karlin

Edited by Michèle Wright

First published in 1985
Sinclair Research Ltd
25 Willis Road, Cambridge CB1 2AQ, England

ISBN 1 85016 036 8

Made in the UK.

Contents

Contents *continued*

1.0 About this Guide

This guide describes the methods which may be used for machine code programming on the QL. Its contents are also relevant to compiler writers who must implement a run-time library for other languages. This guide describes only those techniques which are specific to the QL. It does not contain a general description of 68000 or 68008 assembly language programming: this information can be obtained from a number of different sources, details of which may be found in the bibliography. It is, therefore, **strongly recommended that a reference book describing 68000 assembly language** be consulted before attempting to understand this guide.

The guide also gives details of how various peripherals such as hard disk interfaces, add-on memory and ROM cartridges may be added on to the QL, with many details about how the firmware for such devices should be written.

Readers may notice that there are no circuit diagrams or detailed explanations of the QL's internal hardware structure in this manual. This is because it is not necessary to have such information in order to write software for the QL. We have tried in the design of Qdos to provide you with a stable interface to the machine through its operating system; everything you need is there and so long as you build your products using the interface provided there is no danger that any future upgrade of the QL will introduce an incompatibility with existing software products. We will, in short, continue to support all of the system routines documented in this guide, but specifically reserve the right to change the QL's hardware or firmware in any other way we think fit. Provision of circuit diagrams and the like would, apart from endangering the safety of our design patterns, be giving you a route to build products that rely on nonsupported elements in the QL's design.

The commercial section of this guide sets out the various options offered by Sinclair Research for the distribution of QL Software. Its aim is to give you an idea of the way in which we work and the likely channels through which a potential product would pass before it is accepted for publication and offered for sale to our customers. The section also gives information on the purchase and duplication of Microdrive cartridges.

Finally, should you feel that anything essential is missing from this manual we would be very grateful if you would write and tell us. The address to write to is:

Software Publishing Department
(Technical Manual)
Sinclair Research Limited
25 Willis Road
Cambridge CB1 2AQ

2.0 Introduction to Qdos

Qdos is the QL operating system. It is a single-user multi-tasking operating system: that is, it provides the means for several independent programs to run concurrently in the QL, but does not provide any mechanisms to prevent those programs from interfering with each other. Qdos can be thought of as a collection of several things:

1. A set of useful routines for performing functions such as memory allocation, Input/Output, etc.

2. A mechanism for maintaining lists of things to be done on interrupt, including the function of allocating slots of CPU time to programs which require them.

3. A mechanism for starting up the computer, and determining the configuration of any add-on hardware that is connected to it.

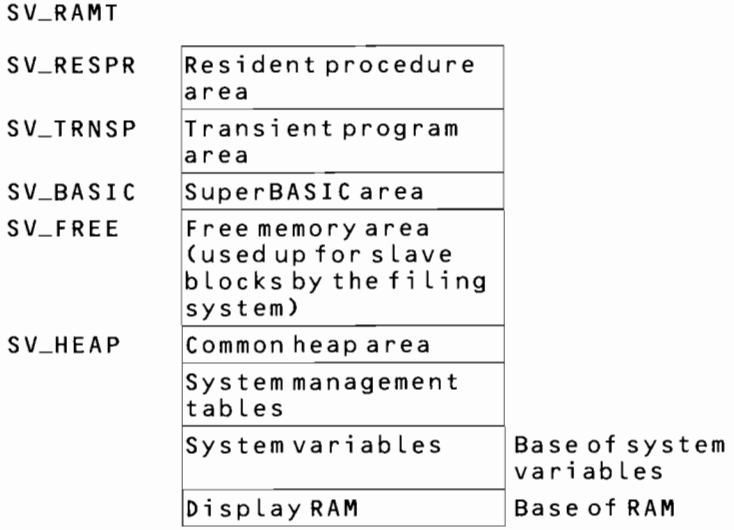
The Qdos mechanisms for start-up are described in section 2.4. Once start-up has been performed, Qdos does not "run" in the sense that traditional operating systems run: its pieces of code and data structures simply exist for programs to use. There is no Qdos "main program" that maintains continuous control of the machine: the SuperBASIC interpreter, which takes the place of the command interpreter found in traditional operating systems, is simply a program which runs on the QL and uses Qdos's facilities, albeit with a number of special provisions. It is possible, and indeed commonly done, to destroy the SuperBASIC interpreter completely, and yet still use all the facilities of the operating system.

Note that in this guide, hex numbers are preceded by a dollar sign ($) as used in the Motorola assembly language format.

2.1 Memory Map

This section describes how Qdos maintains its RAM area. In the QL, the RAM starts with the screen RAM at address $20000, and the area available to Qdos starts at $28000. In an unexpanded QL, the RAM finishes at $3FFFF, whilst in a QL with expansion memory, the RAM may go up as far as $BFFFF. The Qdos initialisation routine determines the amount of RAM present and adjusts the position of its pointers accordingly.

The memory map is as follows:



2.1.1 Principles –

There is no memory management hardware in the QL. This means that all code must execute from fixed addresses in physical memory, and that a piece of code may not be moved after it has been loaded into memory. For this reason, memory is usually allocated in fixed size areas which remain in a fixed location until deleted. The SuperBASIC area is an important exception to this.

2.1.2 System Variables –

The Qdos system variables are a block of memory containing information required by the operating system.

This block is normally located at address $28000, but is not fixed at this address in principle. Applications programs should not rely on that fixed address, but should get the address of the base of system variables by calling the **MT.INF** trap (see section 13.0).

Some of the system variables can usefully be monitored by applications programs, and some of them can safely be altered. A complete list of the system variables, each with its size and offset from the base of system variables, is given in section 18.2.

Included in the system variables area are a set of longword pointers indicating the locations of the other areas in the memory map.

2.1.3 System Management Tables –

Immediately above the system variables are various tables used by Qdos to maintain the list of jobs and various other pieces of information. The supervisor stack also resides in this area.

2.1.4 Common Heap Area –

The common heap area contains the channel definitions which are maintained by the I/O sub-system, together with the working storage required by the I/O drivers or programs. The allocation of space in this area is carried out either by device drivers, when invoked, or directly by jobs. There are two traps provided to allocate and release space in this area: **MT.ALCHP** and **MT.RECHP** (see Section 13.0). The heap allocations of a job are automatically released when the job is removed.

The common heap is an example of the use of a general heap mechanism provided by Qdos, which operates in the way described in the entry for **MT.ALLOC** in section 13.0.

The user code needs to retain one pointer to the free space in the heap. This is a long word and is a relative pointer to the free space in the heap. When the heap has no free space, either because it does not exist, or because it is full, this pointer is zero.

2.1.5 Free Memory Area –

The free memory area is used by Qdos as a buffer memory for the Microdrives, or, if Qdos is suitably extended, for other filing system devices. The area is structured as a collection of *slave blocks*, that is, blocks which are associated with a physical block on a medium. When memory is allocated in another area which would encroach on the free memory area, Qdos must remove one or more slave blocks. Before such a removal takes place, Qdos ensures that a true copy of the information is present on the medium.

Whilst the common heap grows upwards into the free memory area, the areas above it grow downwards into it. As there are three areas above it (the resident procedure area, the transient program area and the SuperBASIC area), special provisions are made so that all three can grow at the appropriate times.

2.1.6 SuperBASIC area –

The SuperBASIC interpreter owns a special area located immediately above the free memory area: this area is used for all the interpreter's storage requirements such as the SuperBASIC program, its variables, its table of I/O channels and the interpreter's working storage. This area is noteworthy in that it can be moved by Qdos without the knowledge of the SuperBASIC interpreter if an area above it needs to grow, or if the SuperBASIC area itself needs to shrink. Its size may also be altered. The mechanism which makes such movement or alteration in size possible operates as follows.

All references to the SuperBASIC area are made relative to the address register A6, and the value of A6 on entry to the interpreter is adjusted by Qdos to the current base of the SuperBASIC area (which is held in the system variable **SV_BASIC**), offset by the length of the interpreter's job header (currently $68 bytes).

The SuperBASIC interpreter divides its working area into several portions, details of which may be found by looking at the **BV** definitions in section 18.3. All of the pointers to these various portions are also relative to A6.

2.1.7 Transient Program Area –

The transient program area is the area of memory into which the user's applications programs are loaded. Each *job* is allocated a block of memory in the transient program area, which it keeps until it is deleted: this area is used for the job's code, data and stack. Programs loaded in this way are not normally re-entrant, but it is relatively straightforward to use the mechanisms in the system to set up a single piece of code which is shared by several different jobs with different data areas.

There is no safe way of determining a priori where a program will be loaded, therefore programs are normally position independent (see section 3.1 on jobs).

2.1.8 Resident Procedure Area –

Memory allocated in this area is unavailable to the operating system. The system knows only two things about the resident procedure area: how to allocate memory in it, and how to release it completely. Both of these operations can only be carried out when there are no transient programs in the machine, due to the fact that the transient program area cannot be moved. Normally, the allocation is done immediately after start-up, and deallocation is never performed.

The area is normally used to load in machine code procedures and functions written to extend the SuperBASIC language (see section 9.7), and occasionally for loading in the code of device drivers when these are not located in ROM in an add-on device.

2.2 Calling Qdos Routines

There are two categories of Qdos routines available to the user: traps and vectored routines. The mechanism for calling a routine is different for each of these two categories.

2.2.1 Traps –

Traps are called using the 68008 TRAP #n instruction: on the QL, this has the effect of a subroutine call to a defined location which has the side effect of saving the status register and entering supervisor mode.

Of the sixteen trap numbers available on the 68008, numbers 0 to 4 inclusive are defined for use by Qdos, the remainder being free for the user to redirect to his own routines. Roughly speaking, the traps are utilised as follows:

TRAP #0 Special trap for entering supervisor mode

TRAP #1 Manager traps – routines which perform overall control of the QL hardware and of the operating system's resources

TRAP #2 Input/Output management traps (I/O traps which allocate resources)

TRAP #3 Input/Output traps which do not allocate resources

TRAP #4 Special trap for the SuperBASIC interpreter.

Traps are called by setting up any required parameters in registers A0–A3 and D1–D3, setting up the code for the required trap in D0 (usually with a MOVEQ instruction), then executing the TRAP instruction. Trap routines do not affect D4 to D7 or A4 to A6. There are, however, a few defined cases which are exceptions to this.

When the TRAP operation is complete, control is returned to the program at the location following the TRAP instruction, with an error key in all 32 bits of D0. This key is set to zero if the operation has been completed successfully, and is set to a negative number for any of the system-defined errors (see section 17.1 for a list of the meanings of the possible error codes). The key may also be set to a positive number, in which case that number is a pointer to an error string, relative to address $8000. The string is in the usual Qdos form of a word giving the length of the string, followed by the characters.

Note that all traps can return the error code **ERR.BP** (for bad parameter). Note also that the condition codes may not be set according to the error code on return from a trap, thus a program wishing to detect an error should execute a TST.L D0 instruction immediately after the TRAP instruction.

Details of all the Qdos traps are given in sections 13.0–15.0.

2.2.2 Vectored Routines –

In addition to the routines accessed by traps, there are several utility routines which are available to the applications program: their addresses are held in a vector table which is located in the ROM starting at address $C0. A vectored routine can be accessed by the following code:

```
MOVE.W   VECTOR_ADDRESS,An
JSR      (An)
```

where **VECTOR_ADDRESS** is the address of the vector table entry, and An is a suitable address register which is not required by the particular routine on entry.

There are some exceptions to this technique: for some vectored routines, the code is:

```
MOVE.W   VECTOR_ADDRESS,An
JSR      $4000(An)
```

The entries in section 16.0 for vectored routines which require this treatment are suitably marked.

There are no general rules covering the handling of errors in vectored routines. Some routines return an error code in D0 in the same way as traps, but others use the technique of returning to one of a set of alternative return addresses. An example is the vectored routine **MD.SECTR**, which returns to the location after the call if there is a "bad medium" error detected, to the address 2 bytes later if there is a "bad sector header" error detected, and to the address 4 bytes later for a correct completion. Thus the correct code to trap these errors would be:

```
MOVE.W   VECTOR_ADDRESS,An
JSR      $4000(An)
BRA.S    BAD_MEDIUM_ERROR
BRA.S    BAD_SECTOR_ERROR

* Code for processing a correct return starts here
   "
   "
   "
```

```
BAD_MEDIUM_ERROR
* Code for processing a bad medium error starts here
    "
    "
    "
BAD_SECTOR_ERROR
* Code for processing a bad sector error starts here
    "
    "
    "
```

Obviously, a similar mechanism can be used with any number of error returns (including zero or one).

Complete details of the vectored routines are given in section 16.0, including information about the behaviour of each routine when an error occurs.

2.2.3 Atomic Actions –

In general, system calls are treated as atomic: while one job is in supervisor mode, no other job in the system can take over the processor. This provides for resource table protection without the need for complex procedures using semaphores. If a job needs to execute some action other than a single system call into which no other job must be allowed to intervene, it should enter supervisor mode before entering the code which performs this action. Supervisor mode is entered using **TRAP #0**. The stack pointer only is changed by this trap.

A job should only use 64 bytes on the supervisor stack, and all of the space used on this stack **must** be released before exiting supervisor mode. In general, there should be nothing on the supervisor stack when a manager trap is made.

Some system calls are only partially atomic, that is, when they have completed their primary function, some other job may gain a share of CPU time before control returns to the calling job. These partially atomic system calls must not be made from a job in supervisor mode. All of the scheduler calls (ie, **TRAP #1** with D0=4, 5, 8, 9, $A, $B) fall into this category, as do all the I/O calls (**TRAP #3**), unless immediate return (timeout<>0) is specified.

A piece of code in supervisor mode can be interrupted by the frame (50/60 Hz) or external interrupts, so care must be taken, when writing interrupt servers, that the system's internal data structure is not modified, directly or indirectly, by system calls. In practice, since interrupt servers tend only to be moving data into or out of queues, this is not a serious limitation.

2.3 Exception Processing

There are three categories of exception traps on the 68008: user traps, traps for software error conditions, and traps for hardware interrupts. There is also one special hardware trap called "bus error", which can be used to trap bad conditions on the address bus: this trap is not supported by the QL hardware.

User traps 0 to 4 inclusive are treated as defined in sections 13.0 through 15.0.

User traps 5 to 15 inclusive, together with the software error traps for "address error", "illegal instruction", "divide by zero", "check array", "trap on overflow", "privilege violation" and "trace" are redirectable by the user on a per-job basis: see the entry for **MT.TRAPV** in section 13.0.

Traps and exception vectors which are not used by Qdos may be redirected through a table which is set up by a particular job.

If a job has set up a table of trap vectors for itself, then that table will automatically be used when that particular job is being executed. The vector tables used by other jobs will not be affected. A job set up by, even if not owned by, a job which has set up a table of trap vectors, will use the same table as that job, until it is redefined.

If the job ID is a negative word, then the table will be set up for the calling job.

The table is in the form of a long word address for each trap or exception. They are in the following order:

$00	address error
$04	illegal instruction
$08	zero divide
$0C	CHK
$10	TRAPV
$14	privilege violation
$18	trace
$1C	interrupt level 7
$20	trap #5
$24	trap #6
$28	trap #7
$2C	trap #8
$30	trap #9
$34	trap #10
$38	trap #11
$3C	trap #12
$40	trap #13
$44	trap #14
$48	trap #15
$4C	end of table

All interrupts on the QL are auto-vectored, therefore there is no treatment of the 68008 vectored interrupt traps. Interrupts generated by the QL internally are level 2 auto-vectors: the interrupt handling mechanism includes the facility for detecting an interrupt on the EXTINTL (external interrupt, active low) line in the QL's expansion port. (See section 11.1 for details of the interface to the relevant hardware.)

It is also possible to generate a level 7 (non-maskable) interrupt: the treatment of this can also be redirected on a per-job basis. Pressing CTRL–ALT–7 on the keyboard generates a level 7 interrupt and also resets all communications with the IPC: a suitable interrupt handler could be written to perform a warm start on the system to allow partial recovery from a crash.

2.4 Start-up

The first thing that Qdos does when the system is reset is to execute a RAM test. This test determines the amount of contiguous RAM present, and if there is any RAM failure, hangs up the machine.

Qdos then initialises the system variables, the system management tables, and the SuperBASIC area.

The address **$C000** is then checked by Qdos for the characteristic longword **$4AFB0001**: if this is found, Qdos links in the SuperBASIC procedures contained in the ROM, prints out the name of the ROM, and performs a **JSR** to its initialisation point (details of the correct format of the **ROM** are found in section 8.0 on ROM device drivers). It is perfectly in order for the code in this ROM to take over the machine completely and never return to the system, for example if another operating system were being booted.

Qdos then does the same for the other ROMs in the expansion slots.

If all of these ROMs return control to Qdos, the next action is to try to open a device driver "BOOT": if this is found, its contents are loaded as a SuperBASIC program and executed. If no device driver "BOOT" has been linked in, Qdos attempts to find a file "MDV1_BOOT" and load and execute its contents as a SuperBASIC program. If both of these attempts fail, Qdos starts up the SuperBASIC interpreter with an empty program memory.

3.0 Machine Code Programming on the QL

Four types of machine code are available to program the QL, each being used to perform quite different operations: *jobs*, *SuperBASIC procedures* and *functions*, *tasks*, and the operating system or extensions to it. Thus there are several differences in both the form in which they are written, and the way in which they are treated by Qdos.

3.1 Jobs

Most application programs written in machine code or compiled code will be in the form of jobs. A job is an entity which has a share of machine resources: it has a *priority* which allows it to claim time-slots of CPU activity, and it has a fixed-size area of memory where data and code can be stored: code normally starts at the bottom of the area, and data at the top. This area is located somewhere in the *transient program area*.

Note that the command interpreter is itself a job but with the exceptional characteristic that its data area is expandable.

A job also has the ability to **own** I/O *channels* or other jobs. There is no protection between jobs under Qdos, so that channels are available for use by all jobs. Ownership simply implies that when the owner of a channel or job is deleted, the owned channel or job is deleted also (this process continues recursively).

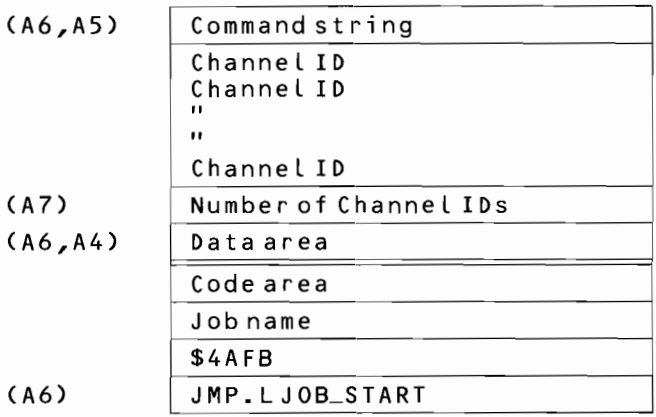
Jobs have three well-defined states: they are *active*, sharing CPU resources with other jobs; *suspended*, for example, waiting for I/O or another job; or *inactive*, occupying memory but not capable of using CPU resources.

The priority of a job can be zero, in which case it is suspended, and does not consume CPU time. It can in fact be suspended for its entire lifetime and never execute at all, which would be the case if it was simply used as a means of obtaining some memory into which data could be loaded. A job at any other priority level is active.

When a job is started, two parts of its area of memory have defined meanings: the bottom of the code area, and the stack, which is at the top of the data area. It is the programmer's responsibility to set up the bottom of the code area, which should be in the following form for use by Qdos utilities:

```
JMP.L  JOB_START
DC.W   $4AFB
DC.W   JOB_NAME_LENGTH
DC.W   'Name of job'
JOB_START
* Code begins execution here (assuming that
the start address defined when the job was
created was zero)
```

On the first occasion that a job is *activated*, (A6) points to the base of the job area, (A6,A4) points to the bottom of the data space, and (A6,A5) points to the top of the job area. There may be some information on the stack, which will be in the following form: (A7) points to the number of channels which have been opened for the job before it was activated; above this is a sequence of longwords holding the channel IDs, and above these are a command string which may have been passed to the job. It is the programmer's responsibility when starting a job to set up this information: the SuperBASIC **EXEC**, **EXEC_W** commands and any utilities produced by Sinclair are compatible with this form.



Note that the normal sequence in Qdos is as follows:

1. reserve space for a job;
2. load its code in;
3. open its channels;
4. activate it.

Execution begins at an address specified when the job was created. This is normally specified as zero, which is why the first thing in a job is normally a **JMP.L** instruction to the entrypoint of the code.

Since Qdos cannot give guarantees as to where a job will be loaded, it is usual to write jobs as position-independent code, although it is possible to avoid this constraint if a special relocating loader is used after the space for the job has been allocated.

The system job table holds information about the jobs within the system. The system variable **SV_JBBAS** points to the base of the job table, and **SV_JBTOP** points to the top. The table is a series of longwords each of which points to a job control block: the contents of this are described in section 18.5. The job is identified to the system by its *Job ID*: this is a longword consisting of a word giving its position in the job table (in the least significant word), and a word of tag allocated by the operating system when the job is created (in the most significant word).

The traps that may be called relating to jobs are as follows:

MT.INF	returns the current job ID, plus miscellaneous information
MT.JINF	returns the status of a job
MT.CJOB	creates a job
MT.JOB	returns information on a job
MT.RJOB	removes an inactive job
MT.FRJOB	forces removal of a job (whether inactive or not)
MT.FREE	finds the largest space available for a job
MT.TRAPV	sets the trap-vector table for a job
MT.SUSJB	suspends a job
MT.RELJB	releases a job
MT.ACTIV	activates a job
MT.PRIOR	changes the priority of a job

A job terminates itself by calling **MT.FRJOB** with its own job ID (or −1, which always refers to the current job).

3.2 SuperBASIC Procedures and Functions

The SuperBASIC command interpreter is job number zero. It behaves like all other jobs in most respects, with the important exception that it owns a special data area which is expandable, and may be moved without the knowledge of the interpreter. This area is located immediately below the transient program area.

Machine code procedures and functions which are added to SuperBASIC appear to the user to be identical to those which are built into the ROM. From the user's point of view they are routines which are executed from within job number zero, but which have certain constraints on the way they are coded.

The most important constraint is that A6 is used to point to the (moveable) base of the SuperBASIC data area. The system may move the area and change the value of A6 between instructions without the knowledge of the interpreter, therefore A6 must not be modified within the procedure or function, and its value must not be stored or used in calculation. This constraint may be side-stepped by entering supervisor mode, but A6 must then be restored on exit back to user mode (the processor is in user mode when a procedure or function is entered). The stackpointer A7 must of course be restored to its original value before exiting from the procedure.

On exit from the procedure, an error key is passed to the interpreter in D0.L: this must be set to zero if there was no error. The procedure or function can then be exited using an **RTS** statement.

If machine code procedures or functions are to be used either recursively or in recursive SuperBASIC procedures, they must obey the usual constraints of having no local variables and no self-modifying code.

Machine code procedures and functions are normally loaded into the *resident procedure area* above the transient program area. This area can only be expanded or deleted when the transient program area is empty, which is normally immediately after the machine is booted.

Trap #4 is the one special trap which relates to SuperBASIC procedures and functions. This trap is used to make the addresses passed to an I/O trap relative to A6, which is necessary when working with the SuperBASIC variables area. It only affects the following trap, and must therefore be called before each trap whose addresses are to be modified.

Details of parameter passing, function returns and other useful information about the SuperBASIC interface are given in section 9.0.

3.3 Tasks

Tasks are special pieces of code invoked under interrupt, usually as part of the physical layer of a device driver. They obey special rules according to the precise conditions under which they are called: these rules are described in the sections on device drivers (sections 6.0–8.0). The important restriction on tasks is that they must not allocate or release machine resources: this should only be done from within a job, or within the access layer of a device driver.

3.4 Operating System Extensions

Some parts of user-defined device drivers do not fit into any of the above categories: they are special routines called from within a job via the Qdos Input/Output sub-system (see section 6.0). These routines have their own rules, and these are described in the sections on device drivers (sections 6.0–8.0).

4.0 Memory Allocation

Memory is allocated differently in each area of the Qdos memory map.

- Memory in the resident procedure area is allocated using the traps **MT.ALRES** and **MT.RERES**.

- Memory in the transient program area is allocated by the mechanisms described in section 13.0 for creation and deletion of jobs. The vectored routines **MM.ALLOC** and **MM.LNKFR** may be used within a job to perform primitive heap allocation inside that job's own data area.

- Memory in the SuperBASIC area is allocated by various mechanisms. The traps **MT.ALBAS** and **MT.REBAS** are used by the interpreter to change the size of the entire area, but are not normally used by anything else. The vectored routine **BV.CHRIX** is used to allocate space on the arithmetic stack: the interpreter itself cleans up this space on return from a procedure or function. Space in the remaining parts of the SuperBASIC area is usually allocated by the vectored routines being used to perform the operations that require the space, so that this allocation is invisible to the user, except that it usually results in a modification of the value of A6.

- Memory in the free memory area is not allocated or deallocated by the user, except by the slave block mechanisms defined in section 7.0 on directory device drivers.

- Memory in the common heap is allocated and released by the traps **MT.ALCHP** and **MT.RECHP**. The area allocated in this way by a job is released when that job is deleted. The same mechanisms can be accessed from within device drivers via the vectored routines **MM.ALCHP** and **MM.RECHP**.

4.1 Heap Mechanism

The mechanisms for allocating and releasing heap space are common to various routines. They are as follows:

A heap is an area of memory which contains a linked list of used heap items, and a linked list of free heap items. Each heap item is an area of memory (which is a multiple of 8 bytes long), together with a pair of longwords: the first is the length of the heap item, while the second is a pointer (relative to itself) to the next heap item in the list. The use of relative pointers ensures that heaps may be moved.

A heap is set up by linking an area of ram→memory into a non-existent heap (free space pointer=0). A heap is expanded by linking an area of ram→memory, preferably but not necessarily, contiguous with the current top of the heap, into the heap.

Provided the user code can remember the length of a heap item, all of the memory in it may be used by the code. On allocation of the heap item, the first long word holds its length, and so, if desired, this may be retained by the user code.

The user code requires to keep one pointer to the first free space item in the heap. This is a long word, and is relative. When the heap has no free space, either because it does not exist, or because it is full, this pointer is zero.

Releasing a heap item adds it to the list of free space items within the heap, and consolidates it with adjacent free spaces where appropriate.

5.0 Input/Output on the QL

A QL program uses I/O by accessing the Qdos. The IOSS in turn accesses the *device driver* for the appropriate device. The device driver is a piece of code which can perform low-level I/O routines for a particular device: that device may correspond to a piece of hardware, such as a serial port, or it may be some notional device occupying a piece of memory, such as a *pipe*, which is a communication channel between jobs.

QL I/O is performed through the IOSS using an I/O *channel*. The applications program *opens* a channel by passing a *device name* to the IOSS, which returns a *channel ID*. The IOSS and the built-in device drivers have the ability to recognize qualifiers appended to the actual name of the device which can direct the open operation in particular ways, such as identifying a file name, or selecting some hardware option. The program then uses the channel ID to identify to the IOSS which channel it wishes to access when performing read or write operations on it. It can also *close* the channel, passing the channel ID to the IOSS. There may be several channels open which use the same device driver, such as multiple screen windows, or Microdrive files. For this reason, all the built-in drivers are re-entrant, as must be the user-defined drivers if they are to have the same capability.

The QL ROM contains drivers for several devices such as screen windows, serial ports, pipes, microdrives, and so on. The user can add his own device drivers for pieces of add-on hardware, or simply for additional functions with the existing hardware.

Note that a channel ID is not the same thing as a SuperBASIC channel number (denoted by #expression): the latter is the index of an entry in the SuperBASIC channel table which includes a channel ID. See sections 18.4 and 18.7 for details of the channel table.

5.1 Serial I/O

All device drivers have, at the very least, the capability to perform serial I/O: that is, the operations of reading bytes, writing bytes, and testing for pending input. Serial I/O is completely byte-oriented – unlike many operating systems there is no inbuilt record structure, which means that the user is free to superpose his own record maintenance in whatever form he wishes. I/O which is purely serial is completely redirectable: when different devices are being used, the device name passed to the channel open trap is the only thing that changes.

The IOSS supports one control character only, this being the *newline* character, which is ASCII 10 ($0A). Whilst this has the disadvantage that one cannot directly store files of graphics commands which can be retrieved by a simple copy, it does have the advantage that files containing arbitrary sequences of bytes cannot do irretrievable damage to the system by being copied to a device for which they were not intended. The serial driver has the option of supporting ASCII 13 as a newline, and ASCII 26 (CTRL-Z) as an end of file marker.

All serial I/O calls support a time-out feature, which may be zero (return immediately), indefinite (wait until the operation is complete), or finite (wait until the operation is complete, or for a set time, whichever is the sooner). This last feature makes it very easy to write code which, for example, puts up a menu only if the user hesitates.

The IOSS supports the following calls for serial I/O:

IO.OPEN	opens a channel
IO.CLOSE	closes a channel
IO.PEND	tests for pending input
IO.FBYTE	fetches a single byte
IO.FLINE	fetches a line of bytes terminated by newline (ASCII 10)
IO.FSTRG	fetches a string of bytes
IO.SBYTE	sends a single byte
IO.SSTRG	sends a string of bytes

The fetch and send traps have several special meanings when used in conjunction with screen or console channels: for a more detailed description of these, see section 15.0 on I/O Traps.

For the fetch byte and fetch string traps, characters read from the keyboard are not echoed in the associated window, and cursor handling is left to the applications program.

5.2 File I/O

Qdos files appear to the applications program as arrays of bytes on a physical device, with an associated *file pointer* which gives the "current position" in a file. A file also has a header, which is normally 64 bytes long containing information about the file such as its name, length, etc. Further details concerning the format of the file header are given in section 7.0 on Directory Device Drivers.

The open call to a file system device supports several modes: old (exclusive), old (shared), or new (exclusive). New (overwrite) mode has a slot allocated in the open keys, but is not currently supported for Microdrives. In addition, a special open key indicates that it is desired to open the directory of the medium for reading rather than a particular file; the directory cannot be explicitly written, but is maintained by the device driver when open calls and deletions are made.

Qdos supports a system of *slaving*, whereby 512-byte blocks of data are buffered in the free memory area (see section 4.0): all unused memory being taken for this area. The filing system may return from a write operation when that operation has only been performed on the slave block concerned; Qdos will later force the system to convert that slave block into a true copy of the data on the physical device. As a result of this mechanism, add-on filing devices normally support 512-byte logical blocks: however this blocking system is transparent to the applications program. A single slave block table is shared by all the directory drivers which want to use it to improve their performance.

In addition to the serial I/O operations described above, Qdos supports the following operations for file-system devices:

IO.FORMT	formats a sectored medium
IO.DELET	deletes a file
FS.CHECK	checks all pending operations on a file
FS.FLUSH	flushes buffers for a file
FS.POSAB	positions the file pointer absolutely
FS.POSRE	positions the file pointer relatively
FS.MDINF	gets information about the mounted medium
FS.HEADS	sets the file header
FS.HEADR	reads the file header
FS.LOAD	loads a file into memory
FS.SAVE	saves a file from memory

The **FS.FLUSH** and **FS.CHECK** command are subtly different: **FS.FLUSH** ensures that all write operations are complete, whereas **FS.CHECK** ensures that all write and read operations (including prefetches) are complete.

5.3 Screen and Console I/O

The keyboard and screen devices are treated in a special way by Qdos, and have a large number of functions in addition to those available for purely serial I/O devices. Two types of device are supported: **scr** (for screen), which is a screen window, and **con** (for console), which is a screen window with an associated keyboard channel. The three channels #0, #1 and #2 which are opened by SuperBASIC are all console channels.

5.3.1 Display Modes –

The QL has two display modes (see the **Concepts** manual for details). The display mode can be set or read using the **MT.DMODE** trap, but as this trap clears all screen windows, it should be used with great care. A program can also find out whether the user selected TV or monitor at switch-on by inspecting the value of the system variable **SV_TVMOD**.

There are two main coordinate systems used for screen I/O: these are the *graphics coordinate system* and the *pixel coordinate system* (see the **Concepts** manual for details). Note that in 256-pixel mode and for several commands in 512-pixel mode, the least significant bit of a dimension in the x-direction is ignored, so that a given pixel address refers to the same location in both modes. Some traps refer to character coordinates: these are based on the pixel coordinate system but are scaled by the current character spacing for the window.

5.3.2 Window Properties and Operations –

A window is an area of screen which may be in any position on the screen, subject to the restriction that its x-position must be an even number. A window may be of any size that does not run off the edge or bottom of the screen, subject to the same restriction. Windows may overlap, but the system does not store or retrieve the area of overlap, it being the user's responsibility to ensure that any information is not lost or garbled.

Each window will have its own particular set of characteristics: a border width, a border colour, a paper colour, a strip colour, an ink colour, a cursor position, a cursor increment, a flag which says whether the cursor is suppressed, a pair of font pointers, information about newline treatment, and graphics information. Details of the window definition block are given in the section 15.0.

The special traps for dealing with windows are as follows:

SD.PXENQ	returns window information in pixel coordinates
SD.CHENQ	returns window information in character coordinates
SD.BORDR	sets the border width and colour
SD.WDEF	redefines a window
SD.CURE	enables the cursor
SD.CURS	suppresses the cursor
SD.SCROL	scrolls a whole window
SD.SCRTP	scrolls the top part of a window
SD.SCRBT	scrolls the bottom part of a window
SD.PAN	pans a whole window
SD.PANLN	pans the line the cursor is on
SD.PANRT	pans the right-hand end of the line the cursor is on
SD.CLEAR	clears a whole window
SD.CLRTP	clears the top part of a window
SD.CLRBT	clears the bottom part of a window
SD.CLRLN	clears the line the cursor is on
SD.CLRRT	clears the right-hand end of the line the cursor is on
SD.RECOL	recolours a window
SD.SETPA	sets the paper colour
SD.SETST	sets the strip colour
SD.SETIN	sets the ink colour
SD.FILL	fills a rectangular block in a window
SD.SETMD	sets the character writing or plotting mode

5.3.3 Screen Character Output Operations –

Newline characters receive slightly different treatment when bytes are being sent to a screen or console channel rather than to any other device. In addition to being caused by a newline character, a newline is automatically inserted when the cursor reaches the right-hand side of the window; when this happens during an **IO.SBYTE** trap, the error code **ERR.OR** (for out of range) is also returned.

If the cursor is suppressed, the newline is held pending. It can be cleared by any call to position the cursor, or activated by any of the following events:

sending another byte or string;
changing the character size;
activating the cursor;
requesting the cursor position.

This feature allows the right-hand character squares to be used without generating stray blank lines.

The following additional operations apply to screen character output:

SD.FOUNT	sets or resets the character fount
SD.SETFL	sets or resets hardware flash (256-pixel mode only)
SD.SETUL	sets or resets underlining
SD.SETSZ	sets the character size and spacing

5.3.4 Graphics Operations –

The QL can perform line, arc or ellipse drawing on a window basis in scaled coordinates. It also provides a primitive area flood routine. The traps are as follows:

SD.POINT	draws a point
SD.LINE	draws a line
SD.ARC	draws an arc
SD.ELIPS	draws an ellipse
SD.SCALE	sets the scale
SD.GCUR	moves the graphics cursor
SD.FLOOD	set or reset area filling

5.3.5 Special Properties of Console Channels –

For the console device, the **IO.FLINE** trap behaves in a particular fashion: the characters typed are echoed in the console window, and the left and right cursor keys (with or without CTRL) are used to edit the line in the standard way. In addition, the cursor is automatically enabled.

An additional trap, **IO.EDLIN**, is provided for console channels, which invokes the line editor on a pre-defined string. The line-editor may be exited by typing **ENTER**, or by typing either the cursor-up or the cursor-down character.

The user can temporarily suspend screen output to a console channel by typing the *freeze screen character* (CTRL-F5). Output is resumed when any character is typed, but the character is ignored for all other purposes. If a non-indefinite time-out has been set for the suspended operation, it may return non-complete if the screen is frozen past the time-out period.

5.3.6 Special Keyboard Functions –

Several console channels may be open at the same time. If they are used by different jobs, it may be that more than one console channel is expecting input at a given time. When this occurs, the user may cycle round the list of console channels currently expecting input by typing the *change queue character* on the keyboard. The cursor in the console window to which keyboard input is currently directed will flash if it is enabled. Any enabled cursors in other windows will be steady.

The change queue character is normally CTRL-C (ASCII 3). It can be changed by modifying the system variable **SV_CQCH**.

The keyboard maintains a type-ahead queue of seven characters in the 8049 processor which controls it. In addition to this, there may be more type-ahead in the queue for each console channel.

The keyboard auto-repeats on all keys except the keyboard change queue character, CTRL-Space (the SuperBASIC break) or CTRL-F5 (the freeze screen character). However, auto-repeat will not occur unless the type-ahead queue for the console channel to which input is currently directed is empty. The delay before auto-repetition begins is held in the system variable **SV_ARDEL**, and the interval between repetitions is held in **SV_ARFRQ** (both in multiples of 1/50th or 1/60th of a second). These can be altered by a program.

When CAPSLOCK is pressed, the system will jump to a user-supplied routine whose absolute address is held in the system variable **SV_CSUB** if the value of this is non-zero. This routine should restore all registers to their initial state before returning.

5.3.7 Extended Operations –

A special trap **SD.EXTOP** is provided to allow a program to invoke a user-supplied routine using the same environment that is passed to the routines in the screen driver. See the description in section 15.0 (I/O Traps) for a more detailed discussion of this trap.

6.0 QDOS Device Drivers

A user-supplied Qdos device driver is a collection of routines which allow an applications program to perform IOSS functions on a user-supplied device in the same way as such functions are performed on the devices built into the system. As these routines are linked into the system's lists in front of the corresponding system routines, they may be used to replace the system routines. At the very least, the device driver contains a set of routines for opening a channel, closing a channel, and performing serial I/O on that channel: these routines are called via the IOSS as part of the job that is performing the I/O. The driver may also include one or more *tasks*, that is, routines performed asynchronously with the calling job, usually under interrupt.

Such tasks, which are known as the *physical layer* of the device driver, normally communicate with the rest of the device driver, which is known as the *access layer*, using asynchronous queues. These queues are usually polled by the task at regular intervals, either on every occasion the scheduler is entered, or on every 50/60 Hz polling interrupt.

Drivers for file system devices use a slightly different, and more general, mechanism: this is described in section 7.0.

Both drivers and tasks are linked in to lists provided by the operating system. The following traps are used to add and remove items from those lists:

MT.LXINT	links in an external interrupt service task
MT.LPOLL	links in a 50/60 Hz polling service task
MT.LSCHD	links in a scheduler loop task
MT.LIOD	links in a device driver to the I/O system
MT.LDD	links in a directory device driver to the file system

MT.RXINT, **MT.RPOLL**, **MT.RSCHD**, **MT.RIOD** and **MT.RDD** remove these links.

The QL provides several utility routines which are useful for various actions commonly performed in device drivers, such as decoding a device name, performing queue operations, etc.

6.1 Device Driver Memory Allocation

Device drivers allocate memory in two areas: the device driver definition block and the channel definition block. The *device driver definition block* belongs to the driver itself, and is allocated by the code which sets up the driver when it is initialised and linked into the various lists. The *channel definition block* belongs to each I/O channel, and is allocated by the driver itself when a channel is opened. Various parts of the channel definition block are thereafter used by the IOSS for its own purposes.

In theory, the access layer can allocate space on the heap at other times: in practice this is not usually required. The whole system can be made re-entrant to allow several channels to be open with the same device driver and the same device driver definition block, but with different channel definition blocks.

Note that the system will certainly crash if the area of a channel definition block is deallocated and used for something else before the channel is closed, or if the area of a device driver definition block is deallocated and used for something else before the device driver is removed from the system's lists, for example if the device driver definition block is in a transient program which is force-removed. This possibility can be obviated by allocating the block in the common heap with a job number of zero, or by allocating it in the resident procedure area.

Tasks must not allocate or release memory: this must be done for them by the access layer, or by the device driver initialisation code.

6.2 Device Driver Initialisation

The code to initialise a device driver must first allocate the space for the device driver definition block, usually by allocating some space in the resident procedure area, although any of the normal allocation mechanisms may be used.

The device driver definition block will normally have the following structure, assuming that A3 has been made to point to it:

$00(A3)	Link to next external interrupt routine
$04(A3)	Address of external interrupt routine
$08(A3)	Link to next poll interrupt routine
$0C(A3)	Address of poll 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)	Any further workspace required for the device driver

The initialisation code should fill in the addresses of the open, close and I/O routines, together with those of any of the routines for tasks that it will be employing. It should also fill in any preset data required in the remainder of the workspace.

Finally, the link routines described above should be called to include the driver in the operating system's lists.

Note that the structure of the first 24 bytes of the device driver definition block is not mandatory; however it is desirable from the point of view of consistency that it be kept the same. The comments in later sections about the base of the device driver definition block being passed to the driver are only valid if the above structure has been used.

6.3 Physical Layer

The physical layer tasks are normally the ones which perform actual I/O under interrupt or polled control. They usually take data out of queues or put data into queues, the other end of such queues being maintained by the access layer.

When the operating system calls one of the tasks in the physical layer, it passes the task a standard set of values in some of the registers. These values are as follows:

D3	Number of 50/60Hz interrupts since last scheduler call (scheduler loop only)
A3	Pointer to base of device driver definition block
A6	Pointer to system variables
A7	Supervisor stack – routines may use up to 64 bytes

6.3.1 External Interrupt Tasks –

An external interrupt task must check its own hardware to determine whether the interrupt was for itself or for some other driver. It may also need to clear the source of the interrupt at that point. If the interrupt was not for itself, it should return.

6.3.2 Polling Interrupt Tasks –

Polling interrupt tasks should only be used when critical timing operations are required. In common with the external interrupt tasks, they can interrupt atomic operations in the rest of the system, such as access layer calls to the same driver, so they should be used with great care.

6.3.3 Scheduler Loop Tasks –

Calls from the scheduler loop do not interrupt atomic tasks. This means that operations such as allocating or releasing memory can be performed safely. Note that it is quite common for the same routine to be included both in the scheduler loop and in the external interrupt list.

Scheduler loop tasks are called at around 50/60Hz when the machine is busy, and more frequently if the machine is idle.

All physical layer calls return with RTS. D0 to D7 and A0 to A6 inclusive may be smashed.

6.4 The Access Layer

The access layer consists of three routines: the channel open, the channel close, and the Input/Output routine. These routines are called for the appropriate driver by the IOSS in response to a user's trap instruction. In the case of the channel open, the routine is called in turn for each device driver in the machine until a driver's open routine returns correctly to indicate that it has recognised the device name. Due to this mechanism, an incorrect open routine may crash the whole system when an open to any device is attempted, whereas the other routines are only invoked in response to the particular device being used.

For all access layer calls, the values of A3, A6 and A7 are the same as for the physical layer. The other registers have different meanings, as described below in the sections for the individual types of call.

All access layer calls return using RTS.

6.4.1 The Channel Open Routine –

When the channel open routine is called via the IOSS, the following registers are set in addition to A3, A6 and A7 which are as described above:

A0 address of the device name
D3 access code as defined in the **IO.OPEN** trap

The open routine should perform the following operations:

First, decode the name; the utility **IO.NAME**, which is described in section 16.0, will normally be used for this purpose. Return with **ERR.NF** in D0 if the name was not recognised by this driver, or with **ERR.BN** if the name was recognised, but some of the additional information was incorrect in value or format.

Then, if the device cannot be shared, check whether the device is in use and prevent another channel from being opened to it. If the device is in use, return **ERR.IU**.

Finally, allocate some space for the channel definition block. Any buffers or working area required for each channel are normally allocated in the common heap. Return with **ERR.OM** if there was not enough memory to do this.

On return from the open routine, the following should be set:

A0 address of channel definition block
A7 stackpointer returned to its value at entry
D0 error return code (zero for a successful open)

The remaining registers may be smashed.

6.4.2 The Channel Close Routine –

When this routine is entered, in addition to the usual values of A3, A6 and A7, A0 points to the base of the channel definition block.

The function of the close routine is simply to release the memory taken up by the channel definition block and to ensure that everything in the device driver definition block is tidy.

Under some circumstances, it may not be possible to close the channel immediately because there are bytes waiting to be transmitted by the physical layer. In this case, the physical layer must contain a scheduler loop task, and the close routine should set a flag for the physical layer to complete the release of the memory on the next invocation of that task in which it is possible to do so. When this happens, it is usually necessary to build in a special mechanism to cope with the undesirable event of a program closing a channel to a particular device, and then re-opening it immediately only to receive an "in use" error because the closed channel has not yet been cleared.

The close routine should return with zero in D0, as it is assumed that a close routine cannot fail. Only registers D0 to D3 and A0 to A3 may be smashed.

6.4.3 The Input/Output Routine –

The I/O routine is called once when an I/O call is made, and then, unless the time-out was set to zero, on every subsequent scheduler loop until the operation is complete or the time-out has expired.

In addition to the usual values of A3, A6 and A7, the following registers are set:

D0	The trap code passed to the IOSS (0 in top three bytes)
D1	Additional information as defined in the trap calls in section 15.0
D2	Additional information as defined in the trap calls in section 15.0
D3	Zero on the first entry for a given trap call, −1 thereafter
A0	Base of channel definition block
A1	Additional information as defined in the trap calls in section 15.0
A2	Additional information as defined in the trap calls in section 15.0

The I/O routine should return **ERR.NC** (not complete) if it cannot complete the operation immediately. If a string operation has been partially completed, the values in D1 and A1 (number of bytes transferred and buffer pointer) should be set appropriately so that the operation can continue on the next try. D0 should be zero on return if the operation has been completed correctly. Registers D2 to D7 may be smashed.

Since most of the code for handling serial I/O is common to all device drivers, the I/O routine usually calls one of the utility routines **IO.SERQ** or **IO.SERIO** (which are described in section 16.0). **IO.SERQ** assumes that the only function of the access layer is to move bytes in and out of a pair of queues pointed to by fixed positions in the channel definition block, while **IO.SERIO** assumes that the operations required of it can all be made up out of three primitive routines for sending one byte, fetching one byte, and checking for pending input, such routines being supplied by the writer of the device driver.

Note that channels are assumed to be bidirectional; it is the responsibility of the I/O routine to trap an operation in a direction that is not allowed.

Note also that output operations which appear to the user as complete have merely completed the access layer call correctly: there being no general way in which the user can ascertain whether the physical layer has in fact completed the operation.

7.0 Directory Device Drivers

Drivers for devices which have a directory and form part of the filing system have a somewhat extended set of functions. For directory device drivers, there are three blocks in which memory is allocated, rather than two: these are the *directory driver linkage block*, the *physical definition block* and the *channel definition block*.

There is one directory driver linkage block for each directory driver: it is an extended form of the device driver definition block as found in a non-directory device driver. The block contains information about how to use the driver, together with the links in the operating system's lists.

Each directory driver may control up to 8 drives (numbered 1 to 8). Each drive has one physical definition block: this contains the drive number and information about the medium.

For each I/O channel that is open, there is an open channel definition block.

The file system is assumed to be composed of 512-byte blocks: thus a byte within a file is addressed by the IOSS by a block number and a byte number within that block. It is of course possible to have a different physical block size, but the mapping of the IOSS structure onto the physical structure will be less convenient.

Each file is assumed to have a 64-byte header (the logical beginning of file is set to byte 64, not byte zero). This header should be formatted as follows:

$00	long	file length
$04	byte	file access key (not yet implemented – currently always zero)
$05	byte	file type
$06	8 bytes	file type-dependent information
$0E	2+36 bytes	file name
$34	long	reserved for update date (not yet implemented)
$38	long	reserved for reference date (not yet implemented)
$3C	long	reserved for backup date (not yet implemented)

The current file types allowed are: 2, which is a relocatable object file; 1, which is an executable program; and 0 which is anything else. In the

case of file type 1, the first longword of type-dependent information holds the default size of the data space for the program.

7.1 Initialisation of a Directory Driver

The initialisation routine should firstly allocate room for the directory driver linkage block, and then write into it the information about the driver routine addresses, the length of the physical definition block required for each drive, and the drive name. Note that for directory drivers, the decoding of the device name is performed by the IOSS, not by the open routine in the device driver as in non-directory drivers: the function of the open routine is to search for the file name within the given drive. The linkage block may be allocated in the resident procedure area if the driver is resident there, but will usually be in the common heap. The system will crash if the linkage block is overwritten without the driver being unlinked.

When this has been done, the traps **MT.LXINT**, **MT.LPOLL**, **MT.LSCHD** and **MT.LDD** can be called to link the driver and any associated tasks into Qdos.

The format of the directory driver linkage block is as follows (assuming that A3 has been made to point to it):

$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-length of drive name characters of drive name (e.g. MDV)

Note that a directory driver must have at least 40 bytes of RAM for the linkage block.

7.2 Access Layer

The access layer of a directory driver contains five routines: the channel open/file delete routine, the close routine, the I/O routine, the forced slaving routine and the format routine.

For all directory device driver access layer calls (including open), A0 points to the base of the channel definition block when each routine is called. However, the format of the block is somewhat different:

The first $18 bytes are reserved for the IOSS.

$18(A0)	**FS_NEXT**	long	link to next file system channel
$1C(A0)	**FS_ACCES**	byte	access mode (D3 on open call, −ve on delete)
$1D(A0)	**FS_DRIVE**	byte	drive ID
$1E(A0)	**FS_FILNR**	word	number of file on drive
$20(A0)	**FS_NBLOK**	word	block number containing next byte
$22(A0)	**FS_NBYTE**	word	next byte from block
$24(A0)	**FS_EBLOK**	word	block number containing byte after EOF
$26(A0)	**FS_EBYTE**	word	byte after EOF
$28(A0)	**FS_CBLOK**	long	pointer to slave block table for current slave block which may hold current/next byte
$2C(A0)	**FS_FNAME**	2+36 bytes	file name
$58(A0)	**FS_SPARE**	72 bytes	spare

A1 points to the physical definition block, which is formatted as follows:

The first $10 bytes are reserved for the IOSS.

$10(A1)	**FS_DRIVR**	long	pointer to access layer link for driver
$14(A1)	**FS_DRIVN**	byte	drive number
$16(A1)	**FS_MNAME**	2+10 bytes	medium name
$22(A1)	**FS_FILES**	byte	number of files open on this medium

7.2.1 The Channel Open/File Delete Routine –

The function of the open routine depends on the access mode. This may have been passed to the IOSS in D3 if the open routine was called as a result of an **IO.OPEN** trap, or it may be a negative number, which would be the case if the routine has been entered as a result of an **IO.DELET** trap.

In order to understand the open routine, it is necessary first to understand the way in which Qdos handles device names. When a device name is passed to the IOSS as a result of an open or delete call, the IOSS looks for a match in its lists of device drivers and directory device drivers. The matching mechanism for non-directory device drivers is defined within the open routine for that driver. The matching mechanism for directory device drivers is as follows. The first characters of the name are checked against the drive name in the directory driver linkage block (which is put there when the driver is initialised), and these are expected to be followed by a drive number between 1 and 8, followed by an underscore, followed usually by the filename. If a match is found, the file system looks to see if there is a physical definition block for that drive already in existence. If there is not, a physical definition block is created in the system's table of physical definition blocks (the drive ID in the channel definition block is an index to this table). Note that the file system has no knowledge of whether a drive is actually connected, and will set up the definition block regardless.

The IOSS then checks to see if this is the second or subsequent open to a shared file: if this is the case it generates the complete channel definition block itself, setting **FS_NBYTE** to $40, and copies the remaining information from the channel definition block for the first open. The directory driver's open routine is not called. Otherwise, the IOSS calls the open routine, passing it the file name in the channel definition block.

On entry to the open routine, the following registers are set:

A0	base of channel definition block
A1	base of physical definition block
A3	base of directory driver linkage block
A6	base of system variables

The channel and physical definition blocks are all set to zero except for the following, which are filled in by the IOSS:

FS_NEXT	link to next file system channel
FS_ACCES	access mode
FS_DRIVE	drive ID
FS_FNAME	file name
FS_DRIVR	pointer to directory driver access layer
FS_FILES	number of files open on this drive (maintained by IOSS)

In the case of a device with removable media, the open routine should find out the name of the medium and install it in **FS_MNAME**. It should also look at the access mode to find out which operation is required. If the required operation is delete, it should perform that operation and return, but if the required operation is another sort of open, then it should fill in the appropriate portions of the channel definition block, namely **FS_FILNR**, **FS_EBLOK**, **FS_EBYTE**,**FS_NBLOK** and **FS_NBYTE**. **FS_CBLOK** is a pointer to the slave block table which may be filled in as an indication to the I/O routine that the block it is looking for may be slaved there. The I/O routine must check this however, normally by searching the slave table.

The IOSS will free the channel definition block on exit from the open routine if the action was a delete or if the open routine returns an error key in D0.

The maintenance of the directory structure of the medium is the responsibility of the open and close routines – the IOSS plays no part in this. Equally, the open routine is responsible for understanding the meaning of the access mode and reacting accordingly.

The open routine may smash registers D1 to D7 and A1 to A5 inclusive before returning. D0 is the error key, and the remaining registers should be preserved.

7.2.2 The Channel Close Routine –

As far as the IOSS is concerned, this routine behaves in the same way as for a non-directory device driver. It is of course necessary for the close routine to maintain the directory structure of the medium, so its operation will normally be rather more complicated.

The close routine for a directory device driver has two additional functions: it must unlink the channel from the list of files open, and must decrement the **FS_FILES** field in the physical definition block, which gives the number of files open on the medium. Suitable code for performing these operations and ending the close routine is as follows:

```
*get address of physical definition block into A2
   MOVEQ    #0,D0              top three bytes must be
                               clear
   MOVE.B   FS_DRIVE(A0),D0    get the drive ID
   LSL.B    #2,D0              convert it to a table
                               offset
   LEA.L    SV_FSDEF(A6),A2    get base of PDB table
   MOVE.L   (A2,D0.W),A2       get address from
                               (base+offset)
*now decrement the file count
   SUBQ.B   #1,FS_FILES(A2)
*now unlink the file
   LEA      FS_NEXT(A0),A0     get address of link
                               pointer . . .
   LEA      SV_FSLST(A6),A1    . . . and pointer to
                               start of linked list
   MOVE.W   UT.UNLNK,A4        routine to unlink an item
   JSR      (A4)
   LEA      -FS.NEXT(A0),A0    restore A0 to base of
                               channel def
   MOVE.W   MM.RECHP,A4        routine to release
                               channel def space
   JMP      (A4)               call it, and exit from the
                               close
```

The close routine must also initiate the process of tidying up any slave blocks remaining for that channel. It need not force the slave blocks to be made into true copies itself, but it must be guaranteed that the copying will happen without further intervention by the calling program.

7.2.3. The Input/Output Routine –

This routine also appears to the IOSS to be identical for both directory and non-directory device drivers, though once again the routine is usually rather more complex for most normal file system devices. The main difference is that the I/O routine for a random access file system device must take into account the current block and position as provided by the IOSS, since these may have been updated by the IOSS as a result of a file pointer positioning trap.

7.3 Slaving

The area of memory between **SV_FREE** and **SV_BASIC** is used by the filing system as temporary storage for file slave blocks and for the slave block table. A slave block is a block of 512 bytes of data. The slave block table is a table of 8 entries whose start point is held in the system variable **SV_BTBAS** and whose top is held in the system variable **SV_BTTOP**; the system variable **SV_BTPNT** points to the most recently allocated slave block table entry. The address of a slave block, relative to the base of system variables, is equal to 512/8 times the offset of the corresponding entry in the slave block table from the beginning of that table.

Currently, only the first byte of each slave block table entry is used by Qdos itself: the remaining bytes are available for use by the driver. This byte is divided into two four-bit nibbles. The most significant nibble contains the drive identifier (0..15), and the least significant nibble is a code indicating the status of the block. The byte is formatted as follows:

$00	unavailable to filing system
$01	empty block
$x3	block is 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

For Microdrives, the remaining space in each slave block table entry is laid out as follows:

BT_PRIOR	01	byte	available for slaving algorithms
BT_SECTR	02	word	physical sector number *2
BT_FILNR	04	word	file number
BT_BLOCK	06	word	block number within the file

It is left to the device driver to decide what the slave blocks are used for but it must be prepared to release a slave block if requested to do so by the memory manager. This is done by calling the driver's forced slaving routine with the following parameters:

A1	points to the base of the offending slave block
A2	points to the physical definition block
A3	points to the base of the directory driver linkage block

Registers D0 to D3 and A0 to A4 inclusive may be smashed. There may not be an error return to this routine.

Typically the slave blocks are used to buffer data being written to a device, the actual writing being carried out by an asynchronous task.

Searching for an empty slave block involves performing a linear search through the slave block table, usually starting from **SV_BTPNT** or **SV_BTBAS**. The status of each entry in the table must be checked and only those blocks which are empty or true representations should be taken. When a new block is allocated **SV_BTPNT** should be updated to point to the allocated block. Allocating slave blocks is a form of memory allocation and should only be carried out by access layer or scheduler loop calls.

This position in memory of a slave block which corresponds to a slave block table entry may be calculated using the following code:

```
    MOVE.L  A4,D0               A4 is pointer to slave
                                block table entry

*
*form offset into slave block table, gives
*slave block no. *8; entries are 8 bytes wide in table
*

    SUB.L   SV_BTBAS(A6),D0
    LSL.L   #6,D0               multiply by 64 (8*64=512)
    MOVE.L  D0,A5
    ADD.L   A6,A5               add offset to system
                                variable base
*A5 now has base address of slave block
```

7.3.1 The Format Routine

This routine is to a large extent independent of the other routines. It is called with the drive number in D1, a pointer to the medium name in A1, and a pointer to the directory driver linkage block in A3.

It should return the error code in D0, the number of good sectors in D1 and the total number of sectors in D2. Registers D3 to D7 and A0 to A5 inclusive may be smashed.

8.0 Built-in Device Drivers

The following devices are built in to the QL ROM:

CON_wXhaxXy_k Console I/O,
window area "w" by "h" pixels, top left hand corner at pixel position "x", "y",
keyboard type-ahead buffer length "k" characters.
The size and position are defined in terms of pixels on a 512×256 display map (position 256×128 is the centre of the screen in both display modes).
Default `CON_448x180a32x16_128`

SCR_wXhaxXy Screen output
window definition is as for CON.
Default `SCR_448x180a32x16`

SERnpz RS232 serial I/O
port "n",
"p" indicates parity: E,O,M,S for even, odd, mark or space parity,
"z" indicates protocol: R indicates raw data,
Z or C indicates that Ctrl-Z is used as an EOF marker,
C indicates that ASCII 13 is to be exchanged with ASCII 10.

Default `SER1R` no parity.

NETI_nn Serial network output
link from node "nn"

NETO_nn Serial network input
link to node "nn"

PIPE_n Job connection and synchronisation
if "n" given it is an output pipe of length n bytes,
otherwise it is an input pipe connected to the channel ID passed in D3.

MDVn_name Microdrive file
MDV1 refers to Microdrive "1".

Within device names, no distinction is made between upper and lower case letters.

9.0 Interfacing to SuperBASIC

When writing SuperBASIC procedures or functions in machine code, there are several things that an applications programmer may want to do: he may wish to look at or modify the information held in SuperBASIC variables and arrays, he may wish to access or modify the SuperBASIC list of I/O channels, and he may wish to reserve and use space on the arithmetic stack. He will also, of course, wish to access the list of parameters passed to the routine and return values either to those parameters or in a function return. In order to do this, it is necessary to understand the data structures used by the interpreter and to emulate the interpreter's techniques for manipulating them.

9.1 Memory Organisation within the SuperBASIC Area

The SuperBASIC area contains twelve distinct areas:

the job header,
the SuperBASIC work area,
the name table,
the name list,
the variable values area,
the channel table,
the arithmetic stack,
the token list,
the line number table,
the program file,
the return list,
the buffer.

There are also various other stacks used by the interpreter.

The job header is located at the bottom of the SuperBASIC area, and looks just like any other job header (see section 18.5). Immediately above this is the SuperBASIC work area; this is an area of fixed storage used for the working variables of the interpreter. Included in these working variables are pointers to the other areas: the interpreter can not only shuffle these areas around, but may also ask Qdos to change the size of the whole SuperBASIC area.

The organisation of this area is shown in section 18.3. Throughout normal operation of the interpreter, A6 points to the base of the SuperBASIC work area, the whole of which may move between instructions, with a corresponding change in A6. All the pointers are, of course, relative to A6, so that their values need not be changed when the SuperBASIC area is moved.

The name table, the name list and the variable values area are required by the applications programmer in order to access and/or modify SuperBASIC variables and parameters. The channel table is required in order to access SuperBASIC I/O channels, and the arithmetic stack (usually abbreviated to RI stack) is a convenient area in which to reserve storage, and is also where parameters are passed. The remaining areas are not described in this document.

9.2 The Name Table

All variables, procedure names, parameters and even expressions are handled through the name table. This is a regular table of eight byte entries, but the entries hold different information according to the type of entry.

The entries may be as follows:

Bytes 7–4	Bytes 3–2	Bytes 1–0	Type
Value pointer	Name pointer	$0001	Unset string
Value pointer	Name pointer	$0002	Unset floating point number
Value pointer	Name pointer	$0003	Unset integer
Ptr to RI stack	−1	$0101	String expression
Ptr to RI stack	−1	$0102	Floating point expression
Ptr to RI stack	−1	$0103	Integer expression
Value pointer	Name pointer	$0201	String
Value pointer	Name pointer	$0202	Floating point number
Value pointer	Name pointer	$0203	Integer
Value pointer	−1	$0300	Substring
Value pointer	Name pointer	$0301	String array
Value pointer	Name pointer	$0302	Floating point array
Value pointer	Name pointer	$0303	Integer array
Line no in msw	Name pointer	$0400	SuperBASIC procedure
Line no in msw	Name pointer	$0501	SuperBASIC string function
Line no in msw	Name pointer	$0502	SuperBASIC f.p. function
Line no in msw	Name pointer	$0503	SuperBASIC integer function
Value pointer	Name pointer	$0602	REPeat loop index
Value pointer	Name pointer	$0702	FOR loop index
Abs. address	Name pointer	$0800	Machine code procedure
Abs. address	Name pointer	$0900	Machine code function

Byte 0 of the name table has an additional usage during parameter passing: see section 9.8.

The Name pointer is a pointer to an entry in the name list (see the following section). A name pointer of −1 indicates a nameless item such as the value of an expression; any other negative pointer indicates a pointer to another entry in the name table of which this entry is a copy.

The Value pointer is a pointer to an entry in the variable values area (see section 9.4). A value pointer of −1 indicates that the value is undefined.

Since all these areas may move during execution, the pointers are offsets from the base of each area. For the RI stack, the base is at the high address; for the others it is at the bottom.

Note that functions written in SuperBASIC are typed according to whether the name ends in **%**,**$** or neither. Functions written in machine code, in common with procedures written in SuperBASIC or machine code, have no type.

The entries for expressions and substrings are for use within the expression evaluator: the applications programmer would not normally use them.

9.3 Name List

The names in the name list are stored as a byte character count followed by the characters of the name. Note that this format is different from all other uses of strings in Qdos in which a word character count is used.

9.4 Variable Values Area

This area is a heap in which the values are stored. The format for each type of data item is given in the following sections.

9.5 Storage Formats

9.5.1. Integer Storage
An integer is a 16-bit two's complement word.

9.5.2 Floating Point Storage
A floating point number is stored as a two-byte exponent followed by a four-byte mantissa.

The most significant four bits of the exponent are zero, whilst the remaining twelve bits are an offset from −$800. The mantissa is two's complement and fractional, with bit 31 of the mantissa representing −1, and bit 30 of the mantissa representing +1/2. There are no implicit bits in the mantissa, so either bit 31 or bit 30 will be set for a normalized number, except in the special case of zero.

The value of the number is thus **mantissa *2 to the power (exponent−$800)**. If the mantissa is viewed as two's complement absolute (as opposed to fractional), the value of the number is given by: **mantissa *2 to the power (exponent−$81F)**. The $1F corresponds to 31 decimal: the length of the mantissa minus one.

Examples of floating point storage are as follows:

Hex		*Decimal*
0804	50000000	10.00
0801	40000000	1.00
07FF	40000000	0.25
07FF	80000000	−0.50
0800	80000000	−1.00
0000	00000000	0.00

9.5.3 String Storage
A string is stored as a word character count, followed by the characters of the string. The string storage always takes a multiple of two bytes. Examples are as follows:

Hex		*String*
0004	41424344	"ABCD"
0003	414243xx	"ABC"
0000		""

9.5.4 Array Storage

An array descriptor has a header which consists of a longword offset of the array values from the base of the variable value area, followed by the number of dimensions (word), followed by a pair of words for each dimension. The first word is the maximum index, the second word is the index multiplier for this dimension.

The storage of floating point and integer arrays is entirely regular. A floating point array takes 6 bytes per element, an integer array 2 bytes per element.

A string array is stored as an array of characters; except that the zeroth element of the final dimension is a word containing the string length. The final dimension defines the maximum length of the string. This is always rounded up to the nearest even number.

A substring is the result of internal slicing operations; this is a regular array of characters; the base of the indexing is one rather than zero.

Examples of Floating Point Storage

Floating point variables (in hex)

0000	0000	0000	0.0
0801	4000	0000	1.0
0800	8000	0000	−1.0
0804	5000	0000	10.0

Floating point arrays

base,2,3,3,2,1	DIM A(3,2)

Examples of String Storage (Numbers are in decimal)

String variable	
4;65,66,67,68	"ABCD"
String array	
base,2,3,12,10,1	DIM A$(3,10)
4;65,66,67,68,x,x,x,x,x,x	"ABCD"
9;49,50,51,52,53,54,55,56,57,x	"123456789"
0;x,x,x,x,x,x,x,x,x,x	""
1;32,x,x,x,x,x,x,x,x,x,	""
Substring array	
base,1,3,1	A$(0,1 TO 3) as above
65,66,67	"ABC"

9.6 Code Restrictions

There is a simple set of rules for writing procedures in machine code for SuperBASIC.

1. As the SuperBASIC program area is liable to move at any time while the execution is in user mode, all refererences to this area must be indexed by A6 or A7. A6 and A7 must never be saved, used in arithmetic or address calculations, and must never be altered, except by pushing or popping the A7 stack. In extreme circumstances it is possible to enter supervisor mode (Trap #0) to make the following action atomic. If this is done, A6 and User stack pointer must not be saved or manipulated before entering supervisor mode, and they must be restored before exiting.
2. Not more than 128 bytes must be used on the user stack.
3. D0 must be returned as an error code (long).
4. D1 to D7 and A0 to A5 inclusive may be treated as volatile.

9.7 Linking in New Procedures and Functions

New SuperBASIC procedures and functions written in machine code may be linked into the name table using the vectored routine **BP.INIT** (see section 16.0). When the procedures and functions are in a ROM in the suitable format (see section 11.4), **BP.INIT** is called automatically. If the procedures and functions are to be stored in RAM, they should be loaded into the resident procedure area as, once added, they may not be removed except by re-booting the machine. It is usually convenient to load the code for calling **BP.INIT** to make the linkage into the same area, although this is not necessary.

9.8 Parameter Passing

The SuperBASIC interpreter passes parameters using a substitution mechanism, which operates as follows. The interpreter first evaluates any of the parameters that are expressions. A new entry is then created at the top of the name table for each actual parameter. In the case of a procedure or function written in SuperBASIC, this is followed by a null entry for any formal parameter that is missing from the actual

parameter list. The interpreter then swaps the new name table entries with the old name table entries corresponding to the actual parameters. In the case of a procedure or function written in machine code, the code is then called with A3 pointing to the name table entry for the first parameter in the list, and A5 pointing to the last ((A5–A3)/8 is the number of parameters).

If a local statement is encountered, the entry in the name table is copied to a new position at the top of the table, and an empty entry put in its place.

At the end of a SuperBASIC procedure or function, the parameter entries are copied back and local variables are removed. The parameter entries in the name table together with any temporary storage in the variable value table are then removed for all procedures and functions.

Byte 0 of the name table entry for a parameter has an additional meaning to that associated with a normal name table entry. The bottom four bits have the usual indication of type (0=null, 1=string etc.), but the top four bits are used to indicate the separator that was present after the parameter in the actual parameter list, together with information as to whether the actual parameter was preceded by a hash (#).

Thus the format of byte 0 is as follows:

h sss tttt

type: 0=null, 1=string, 2=floating point, 3=integer

type of following separator: 0=none, 1=comma, 2=semi-colon, 3=backslash, 4=exclamation mark, 5=TO

1 if the parameter was preceded by hash, otherwise 0

9.9 Getting the Values of Actual Parameters

For the purpose of using scalar (as opposed to array) parameters locally in the same way as "call by value" parameters in other high-level languages, it is expedient to use one of a set of four vectored routines which place the values of actual parameters on the arithmetic stack. Each routine assumes that all the parameters will be of the same type. It is passed the values of A3 and A5 which point to the name table entries for the parameters; it returns the number of parameters fetched

in the least significant word of D3, and the values themselves in order on the arithmetic stack with the first parameter at the top (lowest address) of the stack. These routines smash the separator flags. They are as follows: **CA.GTINT** gets 16-bit integers, **CA.GTFP** gets floating point numbers, **CA.GTSTR** gets strings, and **CA.GTLIN** gets floating point numbers but converts them to 32-bit long integers.

These routines may still be used when processing parameters of mixed type or when wishing to inspect the separators. To begin with, the values of A3 and A5 should be saved; then, for each parameter in succession, the separator flags are inspected, and the appropriate routine is called with A3 pointing to the parameter and A5 equal to A3+8, thus getting one parameter.

These routines smash D1, D2, D4, D6, A0 and A2. The error codes are returned in D0 and the condition codes.

A special technique is provided for use in those routines in which it is necessary for the user to be able to type in a string without quotes, as it's required for SuperBASIC commands involving device names. Firstly, the name is inspected to see if it is a valid set string variable. If it is, the string is fetched using **CA.GTSTR**; if it is not, the parameter's name itself is fetched from the name list, and converted to string form by changing its word count from byte to word, realigning the string if necessary. If a string is to be input without quotes, it must of course follow the rules for SuperBASIC names, as described in the **Concepts** manual.

9.10 The Arithmetic Stack Returned Values

The top of the arithmetic stack is usually pointed to by A1. Space may be allocated on the stack by calling the vectored routine **BV.CHRIX**: the number of bytes required is given in D0.L; D0 to D3 are smashed by the call. Since both the stack within the SuperBASIC area and the SuperBASIC area itself may move during a call, the stack pointer should be saved in **BV_RIP(A6)** before the call, and restored from **BV_RIP(A6)** after the call has been completed. The routine ensures that the restored value will be correct.

The vectored routines for getting parameters reserve their own space on the arithmetic stack.

The arithmetic stack is automatically tidied up both after procedures, and after errors in functions. To make a good return from a function, the returned value should be at the top (lowest address) of the stack with nothing below it (that is with both (A6,A1.L) and **BV_RIP(A6)** pointing to it) when the routine is exited. The type of the returned value should be in D4 (1=string, 2=floating point number, 3=integer). Since SuperBASIC has no long integer type, long integers must be converted to floating point before returning.

Values can also be returned to parameters or, indeed, global variables, by putting the value on the arithmetic stack in the same way, pointing A3 to the appropriate name table entry and calling the vectored routine **BP.LET**. D0 is an error return, and D1, D2, D3, A0, A1 and A2 are smashed. If the actual parameter was an expression, no error will be given, but the value returned will be lost. The type of the returned parameter is determined by the name table entry, and the information on the arithmetic stack must be in the correct form.

Note that strings must be aligned on the arithmetic stack so that the character count is on a word boundary. All entries on the stack must be a multiple of two bytes long, so that a string of odd length has one byte at the end which contains no information.

9.11 The Channel Table

A channel number (#n) is an index to an entry in the SuperBASIC channel table. This is a table of items which are each of length **CH.LENCH** (currently$28) bytes. The base of the table is at **BV_CHBAS(A6)**, and the top is at **BV_CHP(A6)**; thus the base of the entry for channel #n is given by:

(n*CH.LENCH+BV_CHBAS(A6)) (A6)

The format of each table entry is as follows:

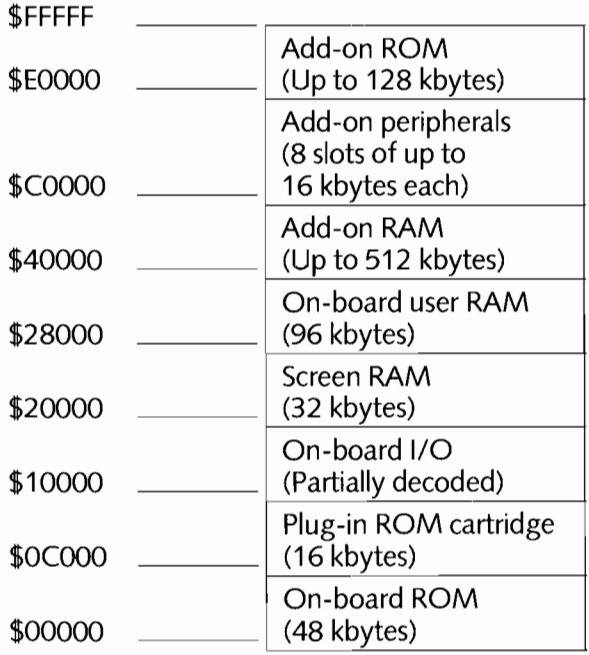
$00	long	the channel ID
$04	float	current graphics cursor (x)
$0A	float	current graphics cursor (y)
$10	float	turtle angle (degrees)
$16	byte	pen status
$20	word	character position on line for PRINT and INPUT
$22	word	WIDTH of page

If a channel entry is off the top of the channel table, or if the channel ID is negative, there is no channel open to that # number.

10.0 Hardware-related Programming

10.1 Memory Map

The 68008 has one megabyte of address space. Although an unexpanded QL uses only the bottom 256 kbytes of this, the allocation for the remainder is determined and should be adhered to when designing add-on hardware. This is how it is made up:



The registers in the on-board I/O area are partially decoded: the details of this decode may vary according to different versions of the QL hardware – some versions will recognise any address in the entire area.

However, the address map normally used is the same for all QLs:

Address (Hex)	Function (Read)	Function (Write)
$18023	Microdrive data (track 2)	Display control
$18022	Microdrive data (track 1)	Microdrive/RS-232-C data
$18021	Interrupt/IPC link status	Interrupt control
$18020	Microdrive/RS-232-C status	Microdrive control
$18003	Real-time clock byte 3	IPC link control
$18002	Real-time clock byte 2	Transmit control
$18001	Real-time clock byte 1	Real-time clock step
$18000	Real-time clock byte 0	Real-time clock reset

The display control registers are in the ZX8301 "Master chip", and the others are in the ZX8302 "Peripheral chip". The details of the QL hardware are rather obscure, and it is strongly recommended that these registers should not be used by applications programs, and should only be accessed via Qdos traps or vectored routines.

10.2 Display Control

The display format in memory is explained below: this format is specific to the QL and may change on future Sinclair products. It is, therefore, strongly advised that screen output be performed using only the standard screen driver, together with the **MT.DMODE** trap.

In 512-pixel mode, two bits per pixel are used, and the GREEN and BLUE signals are tied together, giving a choice of four colours: black, white, green and red. On a monochrome screen, this will translate as a four level greyscale.

In 256-pixel mode, four bits per pixel are used: one bit each for Red, Green and Blue, and one bit for flashing. The flash bit operates as a toggle: when set for the first time, it freezes the background colour at the value set by R, G and B, and starts flashing at the next bit in the line; when set for the second time, it stops flashing. Flashing is always cleared at the beginning of a raster line.

Addressing for display memory starts at the bottom of dynamic RAM and progresses in the order of the raster scan – from left to right and from top to bottom of the picture. Each word in display memory is formatted as follows:

High byte (A0=0)								Low Byte (A0=1)								Mode
D7	D6	D5	D4	D3	D2	D1	D0	D7	D6	D5	D4	D3	D2	D1	D0	
G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	512-pixel
G3	F3	G2	F2	G1	F1	G0	F0	R3	B3	R2	B2	R1	B1	R0	B0	256-pixel

R, G, B and F in the above refer to Red, Green, Blue and Flash. The numbering is such that a binary word appears written as it will appear on the display: ie R0 is the value of Red for the rightmost pixel, that is the last pixel to be shifted out onto the raster.

10.3 Display Control Register

This is a write-only register, which is at $18063 in the QL.

One of its bits is available through the Qdos **MT.DMODE** trap: bit 3, which is 0 for 512-pixel mode and 1 for 256-pixel mode.

The other two bits of the display control register are not supported by Qdos, these being bit 1 of the display control register, which can be used to blank the display completely, and bit 7, which can be used to switch the base of screen memory from $20000 to $28000. Future versions of Qdos may allow the system variables to be initialised at $30000 to take advantage of this dual-screen feature: the present version does not.

Bits 0, 2, 4, 5 and 6 of the display control register should never be set to anything other than zero, as they are reserved and may have unpredictable results in future versions of the QL hardware.

10.4 Keyboard and Sound Control

The keyboard and loudspeaker are controlled by the QL's second processor, which is an 8049 single-chip microcomputer: this is known in the QL as the *Intelligent Peripheral Controller* or *IPC*. The **MT.IPCOM** trap provides a set of commands that the CPU can send to the IPC over the serial link that connects them. This trap is discussed in greater detail in section 13.0.

When the keyboard is accessed via the console driver, the usual functions of debounce and conversion to ASCII are performed, in addition to the functions described in section 15.0. The other way of accessing the keyboard is to use the **MT.IPCOM** trap to monitor the instantaneous state of the keys directly: this is the only way of detecting multiple key presses (necessary for joystick input), or of detecting the state of the SHIFT, CTRL and ALT keys when no other key has been depressed. See the SuperBASIC Keywords entry on the **KEYROW** function for an example of the use of this technique.

The same trap, with different parameters, is used for sound generation.

10.5 Serial I/O

The QL's serial I/O should only be accessed via the serial driver, except for setting the baud rate, which is performed by the **MT.BAUD** trap. The only other function that can safely be performed by the user independently of the operating system is the checking of the transmit handshake lines (DTR on channel 1 and CTS on channel 2), which can be looked at by monitoring bits 4 and 5 of the microdrive status register respectively. Note that if the connector is rewired to use these pins as data lines, this function could be used to perform RS-232-C reception entirely in software, which would make it possible to perform XON–XOFF handshaking or split baud rate operation.

10.6 Real-time Clock

The QL's real-time clock is a 32-bit seconds counter. The three traps **MT.RCLCK**, **MT.SCLCK** and **MT.ACLCK** are used to read, set and adjust the clock. The vectored routines **CN.DATE** and **CN.DAY** are used to convert the time obtained to a string.

10.7 Network

This should not be accessed other than by the built-in device driver.

10.8 Microdrives

Normally, these should not be accessed other than by the built-in device driver. However, it is possible to write routines to access microdrive sectors directly in order to perform such functions as fast medium-to-medium copying or recovery of data from a damaged medium.

There are four vectored routines provided for this purpose: **MD.READ**, **MD.WRITE**, **MD.VERIN** and **MD.SECTR**. Use of these routines requires a detailed understanding of the microdrive hardware and format, and is probably beyond the scope of most users.

However, to use these routines the following code example shows how a microdrive is selected or de-selected. In later versions of the operating system it will be a vectored entry.

```
sys_wser
  move.b d0,-(sp)                          ;save operation
wait
  subq.w #1,sv_timo(a0)                    ;decrement timeout
  blt.s  set_mode                          ;done?
  move.w #(20000*15-82)/36,d0              ;time=18*n+42 cycles
delay1
  dbra   d0,delay1                         ;delay
  bra.s  wait                              ;repeat until timeout
                                           expires
set_mode
  clr.w  sv_timo(a0)                       ;clear wait
  and.b  #pc.notmd,sv_tmode(a0)            ;not RS232
  move.b (sp)+,d0
  or.b   d0,sv_tmode(a0)                   ;either mdv or net
  and.b  #0FFh-pc.maskt,sv_pcint(a0)       ;disable transmit
                                           interrupt
exit
  move.b sv_tmode(a0),pc_tctrel            ;set pc
  rts
sys_rser
  bclr   #pc..serb,sv_tmode(a0)            ;set RS232 mode
  or.b   #pc.maskt,sv_pcint(a0)            ;enable transmit
                                           interrupt
  bra.s  exit
md_desel
  moveq  #pc.desel,d2                      ;clock in deselect bit first
  moveq  #7,d1                             ;deselect all
  bra.s  sedes
```

```
md_selec
  moveq  #pc.selec,d2          ;clock in select bit first
  subq.w #1,d1                 ;and clock it through n
                               ;times
sedes
clk_loop
  move.b d2,(a3)               ;clock high
  moveq  #(18*15-40)/4,d0      ;time=2*n+20 cycles
  ror.l  d0,d0
  bclr   #pc..sclk,d2          ;clock low
  move.b d2,(a3)               ;. . . clocks d2.0 into first
                               ;drive
  moveq  #(18*15-40)/4,d0      ;time=2*n+20 cycles
  ror.l  d0,d0
  moveq  #pc.desel,d2          ;clock high – deselect bit
                               ;next
  dbra   d1,clk_loop
rts
drive
  bsr.s  startup
  bsr.s  wind_dwn
  rts

;  Routine to start up a microdrive.
;  NB: RETURNS IN SUPERVISOR MODE (if d3=1 to 8)

;  d1                          d1 smashed
;  d2                          d2 smashed
;  d3    number of microdrive  d3 preserved
;  a0                          a0 SV_BASE
;  a3                          a3 mdctrl (=18020h)

;  errors:

;  OR: microdrive out of range

startup
  cmp.l  #1,d3                 ;legal microdrive?
  blt.s  ill_drve              ;jump if not
  cmp    #8,d3                 ;legal microdrive?
  bgt.s  ill_drve              ;jump if not
  move.l (sp)+,a3              ;a3=return address
  moveq  #mt.inf,d0            ;select MT.INF
  trap   #1                    ;a0=^to system
                               ;variables
  trap   #0                    ;supervisor mode
  move.l a3,-(sp)              ;'return' (geddit?) the
                               ;return address
  moveq  #10h,d0               ;microdrive mode
  bsr    sys_wser              ;wait for RS232 to
                               ;complete
```

```
  or      #0700h,sr          ;shut out rest of world
  move.l d3,d1               ;d1 is microdrive to be
                             started
  move.l #mdctrl,a3          ;a3=^control register
  bsr     md_selec           ;start it up
  moveq   #0,d0              ;no problems
  rts                        ;return
ill_drve
  moveq   #-4,d0             ;error=out of range
rts

;  Routine to wind down (all!!) microdrives
;  N.B. MUST BE CALLED IN SUPERVISOR MODE

; d1                         d1 smashed
; d2                         d2 smashed
; a0                         a0 SV_BASE
; a3                         a3 ^instruction after
                             call to here(!!)

wind_dwn
  moveq   #mt.inf,d0         ;select MT.INF
  trap    #1                 ;a0=^to system
                             ;variables
  move.l #mdctrl,a3          ;a3=^control register
  bsr.s   md_desel           ;wind it down
  bsr     sys_rser           ;re-enable RS232
  move.l (sp)+,a3            ;a3=return address
  move    #0,sr              ;interrupts off
  move.l a3,-(sp)            ;'return' (it's a killer!)
                             ;return addr.
  rts                        ;return
```

11.0 Adding Peripheral Cards to the QL

Peripheral cards may be plugged into the expansion connector on the left-hand side of the QL, or into one of the connectors in the QL expansion module: a unit which allows several add-on cards to be connected to the QL in parallel. The QL expansion module consists of a power supply and a card cage containing a specially wired backplane. The backplane is connected to the QL via a ribbon cable and buffer card.

There are two general categories of peripheral card for the QL: pure add-on memory cards, and other peripheral cards.

It is intended that only one pure add-on RAM card be plugged into the machine at any one time. It is allocated the address area between $40000 and $BFFFF; the add-on memory should be contiguous from $40000 upwards. This allows for an add-on memory size of up to 512 kbytes.

There is also room for an add-on ROM card of up to 128 kbytes, which is allocated the addresses $E0000 to $FFFFF.

Other peripheral cards contain electronics for the devices being added, a small ROM containing the drivers for the devices being added together with a code allowing the QL to detect that the card is present, and a 4-bit comparator which is used to select the card as explained below.

Note that the convention adopted in this document for an active low signal is to append the letter "L" to the end of the signal name, as in DTACKL, VPAL etc. This takes the place of the overbar indication used in the data sheets from most vendors.

11.1 Expansion Connector

The expansion connector allows extra peripherals to be plugged into the QL. Details of the connections available at the connector may be found in the QL **Concepts** manual.

The connector inside both the QL and the expansion module is a 64-way male DIN-41612 indirect edge connector, as found on standard Eurocard modules. The connector on each add-on card should be the inverse version of this.

The VIN supply is in the region of +9V DC: the trough never falling below 7V. Up to 500 mA may be drawn from this to power the card.

No add-on card should load any pin on the edge connector by more than two LSTTL loads. All add-on card data bus output drivers should be a 74LS245 or equivalent, in terms of drive ability, and being tri-state.

When the expansion module is connected, RESETCPUL is held low until power is applied to the expansion module. Switching off the expansion module also forces RESETCPUL low.

11.2 CPU Interface

The CPU interface is totally memory-mapped onto the 68008's bus, control of the bus for use with the video display controller being obtained by using the DTACKL signal to arbitrate the bus. Memory access is entirely controlled by DSL, with ASL left unused. ASL should not be used to gate any add-on hardware.

An unexpanded QL does not look at address lines A19 and A18. In peripheral cards which are to be added to the QL, it is necessary for each card to disable the circuitry on the QL itself when that peripheral card recognises its own address. This is achieved by pulling signal DSMCL high before DSL goes low including buffering times. This is done typically by using a fast NPN switching transistor (such as an MPS2369) connected as an emitter follower with the emitter connected to DSMCL, the collector to +5V and the base to a logic signal. Note that the timing for this operation is the most critical in most hardware interfaces to the QL, especially when the necessary signals have been buffered.

Add-on cards must supply DTACKL or VPAL as required, to notify the CPU that they have recognised their address.

All 68008 signals are available both on the expansion connector and in the expansion module to allow expansion to include coprocessors or other peripherals.

The following signals are outputs only: A0–A19, RDWL, ASL, DSL, BGL, CLKCPU, E, RED, BLUE, GREEN, CSYNCL, VSYNCH, ROMOEH, FC0–2, RESETCPUL.

The following lines are inputs only, and should only be driven from open collector outputs: DTACKL, BRL, VPAL, IPL0L, IPL1L, BERRL, EXTINTL, DBGL.

The data bus, D0–D7, is bidirectional.

When using the QL expansion module, the data bus buffers in the module are enabled whenever A18 or A19 is high, or if the Data Bus Grab Signal (DBGL) is asserted by any add-on card on pin 25A of the edge connector. If DBGL is to be used, it should be driven by an open collector buffer.

The EXTINTL pin may be used to generate a level 2 external interrupt, which can be linked to a user task (see section 6.3). Note that the EXTINTL pin must not be negated until the Qdos start-up mechanism is complete, or there is a risk of the system hanging up.

11.3 Peripheral Card Addressing

Peripheral cards (other than pure add-on memory cards) are allocated the address space between $C0000H and $DFFFFH. Each peripheral card, when selected, must disable DSMCL and assert VPAL or DTACKL as required, for its own use. This address space is split into eight slots of 16 kbytes each; each peripheral card should normally take only one block if a full set of eight peripheral cards is to be allowed to operate concurrently.

There is a set of four select lines, SP0–SP3, appearing on the edge connector. The first card in the QL expansion module, or a single card directly plugged into the QL, receives a value of zero on these four lines. Each slot in the expansion module has a value one different from that in the other slots: this means that each card is allocated 16 kbytes of address space. The card select logic compares the values on A17–A14 against the number coming in on the select lines in order to determine whether that card is selected. For the card to be selected it must be the case that A14=SP0, A15=SP1, A16=SP2 and A17=SP3.

If there is a ROM containing device drivers for the peripheral card, it should sit in the bottom addresses of the 16 kbyte block. The format of the lowest part of this ROM is specified in the next section.

11.4 Add-on Card ROMs

When the machine is booted, the operating system checks for plug-in ROM drivers by looking for the characteristic longword flag $4AFB0001 at the base of each location in which a ROM might be present. The beginning of a plug-in ROM should be in the following format:

00	$4AFB0001 (flag to indicate ROM is present)
04	pointer to list of BASIC procedures and functions
06	pointer to initialisation routine
08	string identifying the ROM

The pointers are relative to the base of the ROM. If the list pointer is zero then there will be no attempt to link routines into SuperBASIC.

The list of BASIC procedures and functions is in the form used by **BP.INIT** (see section 16.0).

At start-up the machine will link in the additional BASIC procedures from the ROM, then call the initialisation routine (in user mode) which must not modify A6, and finally must restore A0 (the initial window ID), and A3, the pointer to the ROM, on exit. Up to 128 bytes may be used on the user stack.

The description should be in the form of a character count (word) followed by the ASCII characters of the device description(s) ending with the newline character (ASCII 10). It is recommended that the number of characters should be limited to 36.

All code for device drivers must be position independent, since the addresses of the ROM and the devices on the card will be dependent upon the position at which it has been plugged into the QL expansion module. This allows multiple copies of the same add-on card to be used simultaneously.

12.0 Non-English QLs

There are three areas in which non-English QLs may differ from English QLs: the video, the keyboard, and the character set for serial communications.

The version codes for non-English QLs are adjusted appropriately to contain a character identifying the country. In the version code returned by **MT.INF**, this character replaces the decimal point; in the string returned by the SuperBASIC **VER$** function, the character is added on at the end, producing a string three characters long for non-English QLs.

12.1 Video

This is different for countries where the television system is NTSC, which permits the use of fewer raster lines than PAL. In QLs for such countries, the following options are the defaults:

For monitor operation, a 50Hz 624-line non-interlaced system is used; this is the same system as is used on the English QL. The full 512x256 pixel display is available, and the default windows and character size are the same as for the monitor mode on an English QL.

For TV operation, a 60Hz 524-line non-interlaced system is used in which the number of raster lines available is limited to 192. In order to ease the task of software conversion, an alternate display font is provided which allows a 6x8 character square instead of the usual 6x10. This ensures approximately the same number of visible rows of text on both PAL and NTSC QLs, at the cost of true descenders and reduced vertical spacing. The default windows and graphics scaling for TV operation are different from those of the English QL.

It is to be expected that a different version of any applications software (or at least different options) will be required for NTSC operation on domestic televisions.

12.2 Non-English-language Keyboards

The keyboard layout for most European countries will be different from the English layout. This difference should be largely transparent to applications software, since the "QL ASCII" codes contain all the characters necessary for the European countries in question, and the codes generated are independent of the keyboard layout and hence of the actual key depressions required to generate them.

However, there are a few subtleties, the following being the most obvious:

1. A program which draws pictures of keys in certain places will certainly produce an incorrect drawing if the location of those keys has changed between countries.

2. The keyrow function (or **MT.IPCOM** trap) refers to the physical position of the keys, not to their logical meaning. For example, a test on an English QL for the letter "Q" using keyrow will turn into a test for the letter "A" on a French QL which has an AZERTY keyboard.

3. An instruction to "hit any key" will not be strictly accurate for a country which employs non-spacing diacriticals, where the keypress of an accent character does not generate a code until the character to be accented is pressed. The length of the type-ahead buffer in the IPC will be apparently reduced in such cases.

12.3 Character Set

The English character set is available in all countries. However, in non-English countries, the character set for serial communications may (optionally) be translated into a "local" character set, this being chosen by the Sinclair distributor for that country as being a commonly used interface standard. A further option allows the user to specify his own translation table, since it is anticipated that a number of countries will have several standards (i.e., no standards at all).

12.4 Special Alphabets

Languages with non-Roman alphabets, such as Hebrew, Greek, Thai, Arabic, etc., require special treatment. No general scheme has been devised for making software transportable to these countries, and the implementation means will be specific to each country.

13.0 Manager Traps

The special trap #0 is used to enter supervisor mode. The user should store the status register somewhere before calling this trap, so that he can return to user mode by restoring it to its previous value.

TRAP #1 D0=$15 **MT.ACLCK**

Adjust the clock

Call parameters	Return parameters
D1.L adjustment in seconds	D1.L time in seconds
D2	D2 ???
D3	D3 ???
A0	A0 ???
A1	A1 preserved
A2	A2 preserved
A3	A3 preserved

As setting the clock takes a significant time, no adjustment is made if a call is made to adjust the clock and D1=0.

Time starts at 00:00 1 January 1961.

TRAP #1 D0=$A

MT.ACTIV

Activate a job

Call parameters	Return parameters
D1.L job ID	D1.L job ID
D2.B priority (0 to 127)	D2 preserved
D3.W timeout (0 or −1)	D3 preserved
A0	A0 base of job ctrl area
A1	A1 preserved
A2	A2 preserved
A3	A3 preserved if D3=0

Error returns:

NJ job does not exist
NC job already active

This activates a job in the transient area. Execution commences at the start address defined when the job was created.

If the timeout is zero then the execution of the current job continues, otherwise the current job will be suspended until the job activated has completed. The trap will then return with the error code from that job.

TRAP #1 D0=$16

MT.ALBAS

Allocate BASIC program area

Call parameters		Return parameters	
D1.L	number of bytes required	D1.L	nr. bytes allocated
D2		D2	???
D3		D3	???
A0		A0	???
A1		A1	???
A2		A2	???
A3		A3	???
A6	base address	A6	new base address
A7	user stack pointer	A7	new stack pointer

Error returns:

OM out of memory

TRAP #1 D0=$18

MT.ALCHP

Allocate common heap area

Call parameters	Return parameters
D1.L nr. bytes required	D1.L nr. bytes allocated
D2.L owner job ID	D2 ???
D3	D3 ???
A0	A0 base address of area
A1	A1 ???
A2	A2 ???
A3	A3 ???

Error returns:

OM out of memory
NJ job does not exist

This trap is a specific example of the general heap allocation mechanism described in section 2.1.4 and accessible using **MT.ALLOC**.

Trap #1 D0=$C

MT.ALLOC

Allocate an area in a heap

Call parameters		Return parameters	
D1.L	length required	D1.L	length allocated
D2		D2	???
D3		D3	???
A0	ptr to ptr to free space	A0	base of area allocated
A1		A1	???
A2		A2	???
A3		A3	???
A6	base address	A6	preserved

Error returns:

OM no free space large enough

Two trap entries are provided for user heap management where this is required to be atomic. A6 is used as a base address for both this call and for **MT.LNKFR** so that A0 (and A1) is an address relative to A6.

See section 2.1.4 for details of the heap mechanism.

TRAP #1 D0=$E

MT.ALRES

Allocate resident procedure area

Call parameters	Return parameters
D1.L number of bytes reqd.	D1 ???
D2	D2 ???
D3	D3 ???
A0	A0 base address of area
A1	A1 ???
A2	A2 ???
A3	A3 ???

Error returns:

OM out of memory
NC unable to allocate (TRNSP area not empty)

This trap, in common with its partner **MT.RERES** (release resident procedure area) should only be invoked when the transient program area is empty.

TRAP #1 D0=$12

MT.BAUD

Set the baud rate

Call parameters	Return parameters
D1.W baud rate	D1 ???
D2	D2 preserved
D3	D3 preserved
A0	A0 preserved
A1	A1 preserved
A2	A2 preserved
A3	A3 preserved

BP non recognised baud rate

TRAP #1 D0=$1

MT.CJOB

Create a job in transient program area

Call parameters	Return parameters
D1.L owner job ID	D1.L job ID
D2.L length of code (bytes)	D2 preserved
D3.L length of data space	D3 preserved
A0	A0 base of area allocated
A1 start address or 0	A1 preserved
A2	A2 preserved
A3	A3 preserved

Error returns:

OM out of memory
NJ no room in job table or D1 is not a job

This trap allocates space in the transient program area, and sets up a job entry in the scheduler tables. This does not invoke the job and the only initialisation is that two words of 0 are put on the stack. The program itself would normally be loaded, by another job, into the space allocated, after this system call. The stack pointer saved in the job control area points initially to two zero words on the stack (at the highest addresses in the job's data area); if channels are to be opened for the job, or a command string is to be passed to the job, then this can be done before the job is activated.

If D1 is negative, the new job is independent, otherwise it is owned by the calling job.

TRAP #1 D0=$10 **MT.DMODE**

Set or read the display mode

Call parameters		Return parameters	
D1.B key	−1 read mode	D1.B	display mode
	0 mode is 4 colour		
	8 mode is 8 colour		
D2.B key	−1 read display	D2.B	display type
	0 monitor		
	1 625-line TV		
	2 525 line TV		
D3		D3	preserved
A0		A0	preserved
A1		A1	preserved
A2		A2	preserved
A3		A3	preserved
		A4	???

This call is used to set or read the current display mode. It is treated as a manager trap as it affects all the displayed windows. If a call is made to set the screen mode, then all the windows on the screen are cleared and the character sizes may be adjusted. Obviously, there are serious risks involved in calling this trap to set the mode when there are jobs in the machine accessing the screen.

TRAP #1 D0=$6

MT.FREE

Find largest contiguous free space that may be allocated in the transient program area

Call parameters	Return parameters
D1	D1.L length of space found
D2	D2 ???
D3	D3 ???
A0	A0 ???
A1	A1 ???
A2	A2 ???
A3	A3 ???

TRAP #1 D0=$5

MT.FRJOB

Force remove job from transient program area

Call parameters	Return parameters
D1.L job ID	D1 ???
D2	D2 ???
D3.L error code	D3 ???
A0	A0 ???
A1	A1 ???
A2	A2 ???
A3	A3 ???

Error returns:

NJ job does not exist

This inactivates a complete job tree and deletes all jobs in it. If D1 is a negative word then the job is the current job.

Neither of the traps **MT.FRJOB** or **MT.RJOB** to remove jobs can remove job 0.

Neither of these traps are guaranteed atomic.

If there is a job waiting on completion of any job removed, this is released with D0 set to the error code (see **MT.ACTIV** D0=$A).

TRAP #1 D0=$0

MT.INF

System information

Call parameters	Return parameters
D1	D1.L current job ID
D2	D2.L ASCII version (n.nn)
D3	D3 preserved
A0	A0 pointer to system vars
A1	A1 preserved
A2	A2 preserved
A3	A3 preserved

TRAP #1 D0=$11

MT.IPCOM

Send a command to the IPC

Call parameters		Return parameters	
D1		D1.B	return parameter
D2		D2	preserved
D3		D3	preserved
		D5	???
		D7	???
A0		A0	preserved
A1		A1	preserved
A2		A2	preserved
A3	pointer to command	A3	preserved

This trap sends a command to the IPC.

A command sent to the IPC is a nibble followed by a stream of nibbles or bytes being the parameters of the command; some information may then be returned from the IPC. The command format for **MT.IPCOM** is a header describing the command to be sent, followed by the parameters to be sent, followed by a byte indicating whether a reply is expected. The IPC communication is completely unprotected and the command must not contain any errors or else the entire machine will hang up. IPC communications is a very slow process and excessive use of the IPC, for example: polling all rows of the keyboard – the cursor keys have been organised to all be in one row, will cause very high processor overheads.

The command format allows 0, 4 or 8 bits to be transferred from each byte in the parameter block. This is encoded in 2 bits:

00 send least significant 4 bits
01 send nothing
10 send all 8 bits
11 send nothing.

The complete command format is:

1 byte	the IPC command nibble in the LS 4 bits;
1 byte	the number of parameter bytes to follow;
1 long word	containing the codes for the amount of each parameter byte to be sent in reverse order: bits 1,0 the amount of the first byte to send bits 3,2 the amount of the second byte etc.;
n bytes	the parameter bytes
1 byte	length of reply encoded in bits 1,0.

Most of the IPC commands are for use by the operating system and any attempt by application programs to use these is liable to cause loss of data or worse. There are three commands for the IPC which may be used by applications programs:

$9 read a row of the keyboard, 1 parameter
- 4 bits the row number
- 8 bits reply

$A initiate sound, 8 parameters
- 8 bits pitch1
- 8 bits pitch2
- 16 bits interval between steps
- 16 bits duration
- 4 bits step in pitch
- 4 bits wrap
- 4 bits randomness of step
- 4 bits fuzziness
- no reply

$B kill sound, no parameters, no reply.

TRAP #1 D0=$2

MT.JINF

Information on a job

Call parameters	Return parameters
D1.L job ID	D1.L next job in tree
D2.L job at top of tree	D2.L owner job
D3	D3.L MSB −ve if suspended LSB priority
A0	A0 base address of job
A1	A1 ???
A2	A2 preserved
A3	A3 preserved

Error returns:

NJ job does not exist

This trap returns the status of a job.

This trap may be used to check the status of a tree of jobs. On each call D2 should be the ID of the job at the top of the tree; to scan a complete tree the trap is made with D1 being the return value of the previous call. When the tree has been completely scanned D1 is returned equal to zero.

MT.LDD See the entry for **MT.LXINT** for details.

MT.LIOD See the entry for **MT.LXINT** for details.

Trap #1 D0=$D

MT.LNKFR

Link a free space (back) into a heap

Call parameters		Return parameters	
D1.L	length to link in	D1	???
D2		D2	???
D3		D3	???
A0	base of new space	A0	???
A1	ptr to ptr to free space	A1	???
A2		A2	???
A3		A3	???
A6	base address	A6	preserved

A6 is used as a base address for this call and for **MT.ALLOC** so that A0 (and A1) is an address relative to A6.

MT.LPOLL See the entry for **MT.LXINT** for details.
MT.LSCHD See the entry for **MT.LXINT** for details.

TRAP #1 D0=$1A
D0=$1C
D0=$1E
D0=$20
D0=$22

MT.LXINT
MT.LPOLL
MT.LSCHD
MT.LIOD
MT.LDD

Link an external interrupt service routine
a polling 50/60 Hz service routine
a scheduler loop task
an IO device driver
or a directory device driver into the operating system

Call parameters		Return parameters	
D1		D1	preserved
D2		D2	preserved
D3		D3	preserved
A0	address of link	A0	preserved
A1		A1	???
A2		A2	preserved
A3		A3	preserved

TRAP #1 D0=$B

MT.PRIOR

Change job priority

Call parameters	Return parameters
D1.L job ID	D1.L job ID
D2.B priority (0 to 127)	D2 preserved
D3	D3 preserved
A0	A0 base of job ctrl area
A1	A1 preserved
A2	A2 preserved
A3	A3 preserved

Error returns:

NJ job does not exist

This call is used to change the priority of a job. If D1 is a negative word it will change the priority of the current job. Setting the priority to 0 will cause inactivation. This call re-enters the scheduler and so a job setting its own priority to zero will be immediately inactivated.

TRAP #1 D0=$13

MT.RCLCK

Read the clock

Call parameters	Return parameters
D1	D1.L time in seconds
D2	D2 ???
D3	D3 preserved
A0	A0 ???
A1	A1 preserved
A2	A2 preserved
A3	A3 preserved

MT.RDD See the entry for **MT.RXINT** for details.

TRAP #1 D0=$17

MT.REBAS

Release BASIC program area

Call parameters		Return parameters	
D1.L	nr. of bytes to release	D1.L	nr. bytes released
D2		D2	???
D3		D3	???
A0		A0	???
A1		A1	???
A2		A2	???
A3		A3	???
A6	base address	A6	new base address
A7	user stack pointer	A7	new stack pointer

TRAP #1 D0=$19

MT.RECHP

Release common heap area

Call parameters		Return parameters	
D1		D1	???
D2		D2	???
D3		D3	???
A0	base of area to be freed	A0	???
A1		A1	???
A2		A2	???
A3		A3	???

TRAP #1 D0=$9

MT.RELJB

Release a job

Call parameters	Return parameters
D1.L job ID	D1.L job ID
D2	D2 preserved
D3	D3 preserved
A0	A0 base of job ctrl area
A1	A1 preserved
A2	A2 preserved
A3	A3 preserved

Error returns:

NJ not a valid job ID

After this call all jobs are rescheduled.

The activity of jobs can be controlled by activation or by modification of the priority levels. A job at priority level 0 is inactive, at any other priority level it is active.

TRAP #1 D0=$F

MT.RERES

Release resident procedure area

Call parameters	Return parameters
D1	D1 ???
D2	D2 ???
D3	D3 ???
A0	A0 ???
A1	A1 ???
A2	A2 ???
A3	A3 ???

Error returns:

NC unable to release (TRNSP area not empty)

This trap, in common with its partner, **MT.ALRES** (allocate resident procedure area), should only be invoked when the transient program area is empty.

MT.RIOD See the entry for **MT.RXINT** for details.

TRAP #1 D0=$4

MT.RJOB

Remove job from transient program area

Call parameters	Return parameters
D1.L job ID	D1 ???
D2	D2 ???
D3.L error code	D3 ???
A0	A0 ???
A1	A1 ???
A2	A2 ???
A3	A3 ???

Error returns:

NJ job does not exist
NC job not inactive

This trap removes a job (and its subsidiaries) from the transient program area. Only inactive jobs may be removed.

MT.RPOLL See the entry for **MT.RXINT** for details.
MT.RSCHD See the entry for **MT.RXINT** for details.

TRAP #1 D0=$1B
D0=$1D
D0=$1F
D0=$21
D0=$23

MT.RXINT
MT.RPOLL
MT.RSCHD
MT.RIOD
MT.RDD

Remove an external interrupt service routine
a polling 50/60 Hz service routine
a scheduler loop task
an IO device driver
or a directory device driver from the operating system

Call parameters		Return parameters	
D1		D1	preserved
D2		D2	preserved
D3		D3	preserved
A0	address of link	A0	preserved
A1		A1	???
A2		A2	preserved
A3		A3	preserved

TRAP #1 D0=$14

MT.SCLCK

Set the clock

Call parameters	Return parameters
D1.L time in seconds	D1.L time in seconds
D2	D2 ???
D3	D3 ???
A0	A0 ???
A1	A1 preserved
A2	A2 preserved
A3	A3 preserved

TRAP #1 D0=$8

MT.SUSJB

Suspend a job

Call parameters		Return parameters	
D1.L	job ID	D1.L	job ID
D2		D2	preserved
D3.W	timeout period	D3	preserved
A0		A0	base of job ctrl area
A1	address of flag byte	A1	preserved
A2		A2	preserved
A3		A3	preserved

Error returns:

NJ not a valid job ID

A job may be suspended for an indefinite period, or until a given time has elapsed. The timeout period is up to ($7FFF times the frame time).

If the job ID is a negative word, then the current job is suspended. The flag byte is cleared when the job is released. If there is no flag byte, then A1 should be 0. If the timeout period is specified as −1, then the suspension is indefinite; no other negative value should be used. If the job is already suspended, the suspension will be reset. All jobs are rescheduled.

TRAP #1 D0=$7

MT.TRAPV

Set the per-job pointer to trap vectors

Call parameters		Return parameters	
D1.L	job ID	D1.L	job ID
D2		D2	preserved
D3		D3	preserved
A0		A0	base of job
A1	pointer to table	A1	???
A2		A2	preserved
A3		A3	preserved

Note: When a routine in the table is entered as a result of an exception, the CPU is in supervisor mode. The routine should return with an RTE command (not RTS). Any registers used must be saved and restored.

14.0 I/O Management Traps

TRAP #2 D0=$2

IO.CLOSE

Close a channel

Call parameters		Return parameters	
D1		D1	preserved
D2		D2	preserved
D3		D3	preserved
A0	channel ID	A0	???
A1		A1	???
A2		A2	preserved
A3		A3	preserved

Error returns:

NO channel is not open

TRAP #2 D0=$4

IO.DELET

Delete a file

Call parameters		Return parameters	
D1.L	job ID (as file open!!)	D1	???
D2		D2	preserved
D3		D3	???
A0	address of channel name	A0	???
A1		A1	???
A2		A2	preserved
A3		A3	preserved

Error returns:

NO	not opened – too many channels open
OM	out of memory
NF	file or device not found
BN	bad file or device name

TRAP #2 D0=$3

IO.FORMT

Format a sectored medium

Call parameters		Return parameters	
D1		D1.W	number of good sectors
D2		D2.W	total nr of sectors
D3		D3	preserved
A0	ptr to medium name	A0	???
A1		A1	???
A2		A2	preserved
A3		A3	preserved

Error returns:

OM	out of memory
NF	drive not found
IU	drive in use
FF	format failed

The medium name is in the form of a character count (word) followed by the ASCII characters of the drive name, the drive number, underscore then up to 10 characters for the medium name. For example, MDV1_November.

TRAP #2 D0=$1

IO.OPEN

Open a channel

Call parameters		Return parameters	
D1.L	job ID	D1	job ID
D2		D2	preserved
D3.L	code	D3	preserved
	0 old (exclusive) file or device		
	1 old (shared) file		
	2 new (exclusive) file		
	3 new (overwrite) file		
	4 open directory		
A0	address of channel name	A0	channel ID
A1		A1	???
A2		A2	preserved
A3		A3	preserved

Error returns:

NO	not opened – too many channels open
NJ	job does not exist
OM	out of memory
NF	file or device not found
EX	file already exists
IU	file or device in use
BN	bad file or device name

If the job ID is passed as a negative word (for example −1) then the channel will be associated with the current job.

The file or device name should be a string of ASCII characters. This string is preceded by a character count (word), the pointer should point to this word (on a word boundary).

The error return "**BN**" indicates that the name of the device has been recognised but that the additional information is incorrect, for example **CON_512y240**.

The code is usually ignored for access to any non-shared device: in practice, this is anything other than a file store. If the error code is non-zero then no channel has been opened.

Note that New (overwrite) is not currently supported for Microdrive files.

15.0 I/O Traps

TRAP #3 D0=$40 **FS.CHECK**

Check all pending operations on a file

Call parameters	Return parameters
D1	D1 ???
D2	D2 preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved
A3	A3 preserved

Error returns:

NC not complete
NO channel not open

This trap is used to check whether all of the pending operations have completed.

TRAP #3 D0=$41

FS.FLUSH

Flush buffers for this file

Call parameters	Return parameters
D1	D1 ???
D2	D2 preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved
A3	A3 preserved

Error returns:

NC not complete
NO channel not open

When a write operation to a file is complete, the data written may still be in the slave blocks rather than on the file. For further details please see Section 5.2 on File I/O. This call may be used to check that a file is in a known state.

TRAP #3 D0=$47

FS.HEADR

Read file header

Call parameters		Return parameters	
D1		D1.W	length of header read
D2.W	buffer length	D2	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	base of read buffer	A1	top of read buffer
A2		A2	preserved
A3		A3	preserved

Error returns:

NC not complete
NO channel not open
BO buffer overflow

The read header call is provided so that a job can allocate the space for a load call as well as determining the characteristics of a file. The buffer provided must be at least 14 bytes long. In the case of a trap to a pure serial device, then the length of the header returned in D1 will be spurious.

The file pointer is such that position zero is the first byte after the header. Thus block boundaries on standard directory driver files are at positions 512*n-64.

TRAP #3 D0=$46

FS.HEADS

Set file header

Call parameters		Return parameters	
D1		D1.W	length of header set
D2		D2	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	base of header def	A1	end of header def
A2		A2	preserved
A3		A3	preserved

Error returns:

NC not complete
NO channel not open

This call sets the first 14 bytes of the header. The length of file will normally be overwritten by the filing system. When a header is sent over a pure serial device, then the 14 bytes of the header are preceded by a byte $FF.

TRAP #3 D0=$48

FS.LOAD

Load file into memory

Call parameters		Return parameters	
D1		D1	???
D2.L	length of file	D2	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	base address for load	A1	top address after load
A2		A2	preserved
A3		A3	preserved

Error returns:

NO channel not open

Files may be loaded into memory in their entirety with the file load trap. If the transient program area is used for this, a trap #1 must have been invoked to reserve the space before the file load trap is invoked.

D3 should be set to −1 before both this trap, and **FS.SAVE**, and the base address in A1 must be even.

TRAP #3 D0=$45

FS.MDINF

Get information about medium

Call parameters		Return parameters	
D1		D1.L	empty/good sectors
D2		D2	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	ptr to 10 byte buffer	A1	end of medium name
A2		A2	???
A3		A3	???

Error returns:

NC not complete
NO channel not open

The name of the medium, its capacity, and the available space may be obtained for a file or directory that is open.

The medium name is 10 bytes long and left justified. Any remaining bytes are filled with the space character ($20).

The number of empty sectors is in the most significant word (msw) of D1, the total available on the medium is in the least significant word (lsw).

A sector is 512 bytes.

TRAP #3 D0=$42

FS.POSAB

Position file pointer absolute

Call parameters	Return parameters
D1.L file position	D1.L new file position
D2	D2 preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved
A3	A3 preserved

Error returns:

NC not complete
NO channel not open
EF end of file

TRAP #3 D0=$43

FS.POSRE

Position file pointer relative

Call parameters	Return parameters
D1.L offset to file pointer	D1.L new file position
D2	D2 preserved
D3.W timeout	D3 preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved
A3	A3 preserved

Error returns:

NC not complete
NO channel not open
EF end of file

If a file positioning trap returns an off file limits error, then the pointer is set to the nearest limit, this being 0 or end of file. The relative file positioning may, of course, be used to read the current file position.

TRAP #3 D0=$49

FS.SAVE

Save file from memory

Call parameters		Return parameters	
D1		D1	???
D2.L	length of file	D2	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	base address of file	A1	top address of file
A2		A2	preserved
A3		A3	preserved

Error returns:

NO channel not open
DF drive full

In common with **FS.LOAD**, D3 should be set to −1 before this trap, and the base address in A1 must be even.

TRAP #3 D0=$4

IO.EDLIN

Edit a line of characters (console driver only)

Call parameters		Return parameters	
D1	cursor/line length	D1	cursor/line length
D2.W	length of buffer	D2	preserved
D3.W	timeout	D3	preserved
A0	channel ID	A0	preserved
A1	pointer to end of line	A1	pointer to end of line
A2		A2	preserved
A3		A3	preserved

Error returns:

NC not complete
NO channel not open
BO buffer overflow

This is similar to the fetch line trap, except that the pointer A1 is always to the end of the line, D1 contains the current cursor position in the msw and the length of the line in the lsw and the line (from the current cursor position) is written out to the console when the call is made. The line should not have a terminating character when the trap is made, but the terminating character will be included in the character count on return. Enter (ASCII 10), up cursor or down cursor are all acceptable terminating characters.

TRAP #3 D0=$1

IO.FBYTE

Fetch a byte

Call parameters	Return parameters
D1	D1.B byte fetched
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved
A3	A3 preserved

Error returns:

NC not complete
NO channel not open
EF end of file

TRAP #3 D0=$2 or 3

IO.FLINE
IO.FSTRG

D0=$2 IO.FLINE fetch a line of characters terminated by ASCII ‹LF› ($A)
D0=$3 IO.FSTRG fetch a string of bytes

Call parameters		Return parameters	
D1		D1.W	nr. of bytes fetched
D2.W	length of buffer	D2.W	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	base of buffer	A1	updated ptr to buffer
A2		A2	preserved
A3		A3	preserved

Error returns:

NC not complete
NO channel not open
EF end of file
BO buffer overflow (fetch line only)

The character count of a fetch a line trap includes the linefeed character if found.

IO.FSTRG See the entry for **IO.FLINE** for details.

TRAP #3 D0=$0

IO.PEND

Check for pending input

Call parameters	Return parameters
D1	D1 ???
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved
A3	A3 preserved

Error returns:

NC not complete (no pending input)
NO channel not open
EF end of file

This trap is used to check for pending input on a channel. It does not read any data or modify the input channel in any way. This only works on the console device if D3=0 and the keyboard queue is already connected to the console.

TRAP #3 D0=$5

IO.SBYTE

Send a byte

Call parameters	Return parameters
D1.B byte to be sent	D1 ???
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved
A3	A3 preserved

Error returns:

NC not complete
NO channel not open
DF drive full
OR off window/paper etc

TRAP #3 D0=$7

IO.SSTRG

Send a string of bytes

Call parameters	Return parameters
D1	D1.W nr. of bytes sent
D2.W nr of bytes to be sent	D2.W preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1 base of buffer	A1 updated ptr to buffer
A2	A2 preserved
A3	A3 preserved

Error returns:

NC not complete
NO channel not open
DF drive full

Please refer to section 5.3.5 for details of the special treatment afforded to newlines on the console or screen device.
SD.ARC See the entry for **SD.POINT** for details.

TRAP #3 D0=$C

SD.BORDR

Set the border width and colour

Call parameters	Return parameters
D1.B colour	D1 ???
D2.W width	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 preserved
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

This call redefines the border of a window. By default this is of no width. The width of the border is doubled on the vertical edges. The border is inside the window limits. All subsequent screen traps (except this one) use the reduced window size for defining cursor position and window limits.

As a special case, the colour $80 defines a transparent border so that the border contents are not altered by the trap.

If the call changes the width of the border, then the cursor is reset to the home position (top left hand corner).

SD.CHENQ See the entry for **SD.PXENQ** for details.

TRAP #3 D0=$20 to 24

SD.CLEAR

SD.CLRBT
SD.CLRLN
SD.CLRRT
SD.CLRTP

Clear part or all of a window

D0=$20	SD.CLEAR	clear all of window
D0=$21	SD.CLRTP	clear top of window
D0=$22	SD.CLRBT	clear bottom of window
D0=$23	SD.CLRLN	clear cursor line
D0=$24	SD.CLRRT	clear right hand end of cursor line

Call parameters	Return parameters
D1	D1 ???
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

The clear window traps can clear all or part of a window. To clear a part of a window the cursor is used as a reference. The clear operation consists of overwriting all the pixels in the designated area with paper colour.

The division between the top of the window and the bottom of the window is the cursor line. The cursor line is in neither the top nor the bottom of the window.

The cursor line is the whole height of the current character fount (either 10 or 20 rows). The right hand end includes the character at the current cursor position.

SD.CLRBT See the entry for **SD.CLEAR** for details.

SD.CLRLN See the entry for **SD.CLEAR** for details.

SD.CLRRT See the entry for **SD.CLEAR** for details.

SD.CLRTP See the entry for **SD.CLEAR** for details.

TRAP #3 D0=$E

SD.CURE

Enable the cursor

Call parameters	Return parameters
D1	D1 ???
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

The cursor is automatically enabled when a read line or edit line trap is issued to a console window.

TRAP #3 D0=$F

SD.CURS

Suppress the cursor

Call parameters	Return parameters
D1	D1 ???
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

The calls to suppress or enable the cursor do not return an error if the cursor is already suppressed or enabled (respectively), as they merely ensure that the cursor is in the desired state.

SD.ELIPS See the entry for **SD.POINT** for details.

TRAP #3 D0=$9

SD.EXTOP

Call an extended operation

Call parameters		Return parameters	
D1	parameter	D1	parameter
D2	parameter	D2	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	parameter	A1	parameter
A2	start address of routine	A2	preserved

Error returns:

NC not complete
NO channel not open
and anything from the operation routine

This trap invokes an externally supplied routine as if it were part of the standard screen driver. D1, D2 and A1 are passed to the routine, while only D1 and A1 are returned. The code within the routine is executed in supervisor mode with A0 pointing to the channel definition block (see Section 7.2) and A6 pointing to the system variables as for standard device drivers.

TRAP #3 D0=$2E

SD.FILL

Fill rectangular block in window

Call parameters		Return parameters	
D1.B	colour	D1	???
D2		D2.L	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	base of block definition	A1	???
A2		A2	preserved

Error returns:

NC not complete
NO channel not open
OR block falls outside window

This trap fills a rectangular block of a window with the current ink colour, taking into account the mode set by **SD.SETMD**.

The block definition is in the same form as the window definition. It is 4 words long: width, height, X origin and Y origin. The origin is referred to the window origin.

This is a fast way of drawing horizontal or vertical lines.

TRAP #3 D0=$35

SD.FLOOD

Turn area flood on and off

Call parameters		Return parameters	
D1.L	key 0=end flood 1=start or restart flood	D1	???
D2		D2	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1		A1	???
A2		A2	preserved
A3		A3	preserved

Error returns:

NC not complete
NO channel not open

TRAP #3 D0=$25

SD.FOUNT

Set or reset the fount

Call parameters		Return parameters	
D1		D1	???
D2		D2.L	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	base of fount	A1	???
A2	base of second fount	A2	preserved

Error returns:

NC not complete
NO channel not open

The character fount is a 5x9 array of pixels in a 6x10 rectangle. A default fount and a second fount are built into the ROM, although alternative founts may be selected.

If the fount address is given as zero the default fount will be used.

The structure of the fount assumes that up to a certain value characters are invalid (default $1E), from the next value (default $1F) a known number of characters are valid (default $61). Thus the structure is as follows:

$00 lowest valid character (byte)
$01 number of valid characters−1 (byte)
$02 to $0A 9 bytes of pixels for the first valid character
$0B to $13 etc.

Each byte of pixels has the pixels in bit 6 to bit 2 (inclusive) of the byte. The top row of any character is implicitly blank.

If a character, which is to be written, is found to be invalid in the first fount, it is written using the second fount. If it is also invalid in the second fount, then the lowest valid character of the second fount is used.

The default fount extends from $20 to $7F.

SD.GCUR See the entry for **SD.POINT** for details.

SD.LINE See the entry for **SD.POINT** for details.

SD.NCOL See the entry for **SD.POS** for details.

SD.NL See the entry for **SD.POS** for details.

SD.NROW See the entry for **SD.POS** for details.

TRAP #3 D0=$1B, 1E and 1F

SD.PAN
SD.PANLN
SD.PANRT

Pan part or all of a window

D0=$1B	SD.PAN	pan all of window
D0=$1E	SD.PANLN	pan cursor line
D0=$1F	SD.PANRT	pan right hand end of cursor line

Call parameters	Return parameters
D1.W distance to pan	D1 ???
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

The whole of a window, or the whole of the cursor line, or the right hand end of the cursor line may be panned by any number of pixels to the right or to the left. A positive distance implies that the pixels will move to the right. The space left behind will be filled with paper colour.

The cursor line is the whole height of the current character fount (either 10 or 20 rows). The right hand end includes the character at the current cursor position.

SD.PANLN See the entry for **SD.PAN** for details.

SD.PANRT See the entry for **SD.PAN** for details.

SD.PCOL See the entry for **SD.POS** for details.

TRAP #3 D0=$17 **SD.PIXP**

Position cursor using pixel coordinates

Call parameters		Return parameters	
D1.W	X-coordinate	D1	???
D2.W	Y-coordinate	D2.L	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1		A1	???
A2		A2	preserved

Error returns:

NC not complete
NO channel not open
OR off window

The cursor position is the top left hand corner of the next character rectangle referred to the top left hand corner of the window. This trap clears the pending newline in the window.

TRAP #3	D0=$30	SD.POINT	**SD.POINT**
	D0=$31	SD.LINE	**SD.ARC**
	D0=$32	SD.ARC	**SD.ELIPS**
	D0=$33	SD.ELIPS	**SD.GCUR**
	D0=$34	SD.SCALE	**SD.LINE**
	D0=$36	SD.GCUR	**SD.SCALE**

Plot a point, line, arc, ellipse, set scale or graphics cursor position. Expects parameters on the arithmetic stack pointed to by (A1)

Call parameters		Return parameters	
D1		D1	???
D2		D2.L	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	arithmetic stack pointer	A1	???
A2		A2	preserved

Error returns:

NC not complete
NO channel not open

These four traps draw various lines and arcs in the window. Any point on these lines which falls outside the window will not be plotted.

The format of the parameters required is as follows:

SD.POINT	$00(A1)	y-coordinate
	$06(A1)	x-coordinate
SD.LINE	$00(A1)	y-coord of finish of line
	$06(A1)	x-coord of finish of line
	$0C(A1)	y-coord of start of line
	$12(A1)	x-coord of start of line

SD.ARC	$00(A1)	angle subtended by arc
	$06(A1)	y-coord of finish of line
	$0C(A1)	x-coord of finish of line
	$12(A1)	y-coord of start of line
	$18(A1)	x-coord of start of line
SD.ELIPS	$00(A1)	rotation angle
	$06(A1)	radius of ellipse
	$0C(A1)	eccentricity of ellipse
	$12(A1)	y-coord of centre
	$18(A1)	x-coord of centre
SD.SCALE	$00(A1)	y position of bottom line of window
	$06(A1)	x position of left hand pixel of window
	$0C(A1)	length of Y axis (height of window)
SD.GCUR	$00(A1)	graphics x-coordinate
	$06(A1)	graphics y-coordinate
	$0C(A1)	pixel offset to right
	$12(A1)	pixel offset down

For all the graphics traps, the parameters on the A1 stack are floating point, and coordinates are referred to an arbitrary origin (default is 0,0) with an arbitrary scale (default is height of window = 100 units).

The calling program must allocate at least 240 bytes on the A1 stack.

TRAP #3 D0=$10 to 16

SD.POS

SD.NCOL
SD.NL
SD.NROW
SD.PCOL
SD.PROW
SD.TAB

Cursor positioning by character intervals

D0=$10	SD.POS	absolute position
D0=$11	SD.TAB	tabulate
D0=$12	SD.NL	newline
D0=$13	SD.PCOL	previous column
D0=$14	SD.NCOL	next column
D0=$15	SD.PROW	previous row
D0=$16	SD.NROW	next row

Call parameters	Return parameters
D1.W column number (D0=10,11)	D1 ???
D2.W row number (D0=10)	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open
OR position would be out of window

In the case of an error return, the cursor position is not changed. The cursor position is the top left hand corner of the next character rectangle referred to the top left hand corner of the window. These traps clear the pending newline in the window.

SD.PROW See the entry for **SD.POS** for details.

TRAP #3 D0=$A or B

SD.PXENQ
SD.CHENQ

Return the current window size and cursor position

D0=$A SD.PXENQ enquiry in pixel coordinates
D0=$B SD.CHENQ enquiry in character coordinates

Call parameters		Return parameters	
D1		D1	preserved
D2		D2	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	base of enquiry block	A1	???
A2		A2	preserved

Error returns:

NC not complete
NO channel not open

The window size (X,Y) and cursor position (X,Y) are put into a 4 word enquiry block. The top left hand corner of the window is cursor position 0,0. These traps activate the newline if pending in the window.

TRAP #3 D0=$26

SD.RECOL

Recolour a window

Call parameters		Return parameters	
D1		D1	???
D2		D2.L	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	pointer to colour list	A1	???
A2		A2	preserved

Error returns:

NC not complete
NO channel not open

A window may be recoloured without changing the information in it. This allows the same sort of effects as resetting the attributes of an attribute based screen, but it is very much slower.

The colour list is 8 bytes long and should contain the new colours required for each of the 8 colours in the window. Each of the new colours must be in the range 0 to 7. For 4 colour mode, only bytes 0, 2, 4 and 6 need to be filled in.

SD.SCALE See the entry for **SD.POINT** for details.

SD.SCRBT See the entry for **SD.SCROL** for details.

TRAP #3 D0=$18 to 1A

SD.SCROL

SD.SCRBT
SD.SCRTP

Scroll part or all of a window

D0=$18	SD.SCROL	scroll all of window
D0=$19	SD.SCRTP	scroll top of window
D0=$1A	SD.SCRBT	scroll bottom of window

Call parameters	Return parameters
D1.W distance to scroll	D1 ???
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

Part or all of a window may be scrolled; for partial scrolling the cursor is used as a reference. These traps cause pixels to be transferred from one row to another. Vacated rows of pixels are filled with paper colour. A positive scroll distance implies that the pixels in the window will be moved in a positive direction, ie, downwards. The space left behind will be filled with paper colour.

The division between the top of the window and the bottom of the window is the cursor line. The cursor line is included in neither the top nor the bottom of the window. The cursor is not moved.

SD.SCRTP See the entry for **SD.SCROL** for details.

TRAP #3 D0=$2A and 2B

SD.SETFL
SD.SETUL

Set flash and underscore

D0=$2A	SD.SETFL	set flash
D0=$2B	SD.SETUL	set underscore

Call parameters	Return parameters
D1.B 0 attribute off else attribute on	D1 ???
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

SD.SETIN See the entry for **SD.SETPA** for details.

TRAP #3 D0=$2C

SD.SETMD

Set the character writing or plotting mode

Call parameters	Return parameters
D1.W mode	D1 ???
--1 ink is exclusive ored into the background	
0 character background is strip colour	
1 character background is transparent	
0 or 1 plotting is in ink colour	
D2	D2.L preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

TRAP #3 D0=$27 to 29

SD.SETPA
SD.SETIN
SD.SETST

Set screen colours

D0=$27	SD.SETPA	set paper colour
D0=$28	SD.SETST	set strip colour
D0=$29	SD.SETIN	set ink colour

Call parameters	Return parameters
D1.B colour	D1 preserved
D2	D2 preserved
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

The screen driver uses three colours. There is the background colour of a window: referred to as paper colour; this is the colour which is used by the scroll, pan and clear operations. There is the colour which is used by the character generator to provide a highlighting background for individual characters or words: referred to as strip colour. Finally, there is the colour used for writing characters and drawing graphics: referred to as ink colour.

SD.SETST See the entry for **SD.SETPA** for details.

TRAP #3 D0=$2D

SD.SETST

Set character size and spacing

Call parameters	Return parameters
D1.W character width/spacing	D1 ???
0 single width, 6 pixel spacing	
1 single width, 8 pixel spacing	
2 double width, 12 pixel spacing	
3 double width, 16 pixel spacing	
D2.W character height/spacing	D2.L preserved
0 single height, 10 pixel spacing	
1 double height, 20 pixel spacing	
D3.W timeout	D3.L preserved
A0 channel ID	A0 preserved
A1	A1 ???
A2	A2 preserved

Error returns:

NC not complete
NO channel not open

The character generator supports two widths and two heights of character. In 8 colour mode, only the double width characters may be used. In addition the spacing between characters is entirely flexible, but for simplicity of use only two additional spacings are supported directly: these are 8 pixel and 16 pixel, in single and double width respectively.

Calls with D1=0 or 1 in 8 colour mode will operate as though a call had been made with D1 equal to 2 or 3.

SD.SETUL See the entry for **SD.SETFL** for details.

SD.TAB See the entry for **SD.POS** for details.

TRAP #3 D0=$D

SD.WDEF

Redefine a window

Call parameters		Return parameters	
D1.B	border colour	D1	???
D2.W	border width	D2.L	preserved
D3.W	timeout	D3.L	preserved
A0	channel ID	A0	preserved
A1	base of window block	A1	???
A2		A2	preserved

Error returns:

NC not complete
NO channel not open
OR window does not fit on page

This call redefines the shape or position of a window: the contents are not moved or modified, but the cursor is repositioned at the top left hand corner of the new window. The window block is 4 words long and is the width, height, X origin and Y origin.

16.0 Vectored Routines

Vector $110	**BP.INIT**

All addresses passed to this routine must be relative to A6.

BP.INIT is used to link in a list of procedures and functions to be added to the SuperBASIC name table. Once added, the functions can be called from SuperBASIC in the same way as the procedures and functions built into the ROM.

On entry, A1 should point to a table in the following form:

```
        word      approximate number of procedures (see below)

for each procedure
        (word     pointer to routine – here
        (byte     length of name of procedure
        (characters

        word      0
        word      approximate number of functions (see below)

for each function
        (word     pointer to routine – here
        (byte     length of name of function
        (characters

        word      0
```

The "approximate number" of procedures or functions is used to reserve internal table space. It should be exactly equal to the number of procedures or functions unless the average length of the procedure or function names exceeds 7, in which case it should be:

(total number of characters + number of functions or procedures + 7) /8

The pointers to the routines are relative to the address of the pointer (e.g. DC.W ENTRY−*)

All registers except A1, and D2 are preserved by **BP.INIT** and no more than 48 bytes are used on the user stack.

Vector $120 **BP.LET**

All addresses passed to this routine must be relative to A6.

BP.LET assigns a value to be associated with an entry in the SuperBASIC name table. On entry, (A6,A3) should point to the name table entry, and (A6,A1) should point to the value to be assigned (see section 9.5 for details of the storage format for the various types of data). A1 and A3 should be on word boundaries.

The type of the entity to be assigned (and hence its length) is determined by the type in the name table entry.

On exit, D0 is an error code, and D1, D2, D3, A0, A1 and A2 are smashed.

Vector $11A **BV.CHRIX**

All addresses passed to this routine must be relative to A6.

BV.CHRIX is used to reserve space on the arithmetic stack (A6,A1). On entry, the number of bytes required should be in D0.L: D0 to D3 are smashed.

Since not only the stack but the whole SuperBASIC area may move during the call, the arithmetic stack pointer should be saved in **BV_RIP(A6)**, whence it should be retrieved after the call has been completed.

CA.GTFP See the entry for **CA.GTINT** for details.

Vector $112	**CA.GTINT**
$114	**CA.GTFP**
$116	**CA.GTSTR**
$118	**CA.GTLIN**

All addresses passed to these routines must be relative to A6.

These routines are used to get the values of actual parameters to SuperBASIC procedures or functions onto the arithmetic stack. Each routine assumes that all the parameters will be of the same type, as follows:

CA.GTINT 16-bit integer
CA.GTFP floating point
CA.GTSTR string
CA.GTLIN floating point: convert to 32-bit long integer

On entry, (A6,A3) points to the name table entry for the first parameter in the list, and (A6,A5) points to the entry for the last.

The number of parameters fetched is returned in the least significant word of D3. The values themselves are returned in order on the arithmetic stack (A6,A1) with the first parameter at the top (lowest address) of the stack.

These routines smash D1, D2, D4, D6, A0 and A2. D0, and also the condition codes, give the error code. The separator flags in the name table entries are also smashed.

CA.GTLIN See the entry for **CA.GTINT** for details.

CA.GTSTR See the entry for **CA.GTINT** for details.

CN.BTOIB See the entry for **CN.DTOF** for details.

CN.BTOIL See the entry for **CN.DTOF** for details.

CN.BTOIW See the entry for **CN.DTOF** for details.

Vector $EC CN.DATE get date and time
$EE CN.DAY get day of week

CN.DATE
CN.DAY

Call parameters		Return parameters	
D1.L	date (internal value)	D1	preserved
D2		D2	preserved
D3		D3	preserved
A0		A0	preserved
A1	pointer to stack	A1	pointer to stack
A2		A2	preserved
A3		A3	preserved

All addresses passed to these routines must be relative to A6.

There are two date conversion routines: **CN.DATE** returns the date in the form yyyy mmm dd hh:mm:ss, **CN.DAY** returns a three letter day of the week. The result is put on the A1 stack in string format. At least 22 bytes are required by **CN.DATE** and at least 6 bytes by **CN.DAY**.

CN.DAY See the entry for **CN.DATE** for details.

Vector			**CN.DTOF**
	$100 CN.DTOF	floating point	
	$102 CN.DTOI	integer	**CN.BTOIL**
	&$104 CN.BTOIB	binary (byte)	**CN.BTOIW**
	&$106 CN.BTOIW	binary (word)	**CN.DTOI**
	&$108 CN.BTOIL	binary (long)	**CN.HTOIB**
	&$10A CN.HTOIB	hex (byte)	**CN.HTOIL**
	&$10C CN.HTOIW	hex (word)	**CN.HTOIW**
	&$10E CN.HTOIL	hex (long)	

Call parameters		Return parameters	
D1		D1	???
D2		D2	???
D3		D3	???
D7	0 or ptr to end buffer	D7	preserved
A0	pointer to buffer	A0	pointer to buffer
A1	pointer to stack	A1	pointer to stack
A2		A2	???
A3		A3	???

Error returns:

XP error in conversion (eg 1..0 as floating pt.
or no digits or too many hex or binary digits)

All addresses passed to these routines must be relative to A6.

Utilities marked **&** are non-functioning in V1.03 and earlier.

These routines convert from ASCII characters in a buffer (pointed to by A0) to a value on the stack (pointed to by A1).

The hex and binary conversions from ASCII to number, always put a long word on the A1 stack. A1 is set to point to the least significant byte or less significant word for the byte and word conversions.

The decimal conversions may use up to about 30 bytes on the A1 stack.

If there is an error then A0 and A1 are both unchanged. Otherwise, on return, A1 points to the return value (floating point, long word, word or byte) and A0 points to the next character in the buffer.

CN.DTOI See the entry for **CN.DTOF** for details

Vector			**CN.FTOD**
	$F0 CN.FTOD	floating point	
	$F2 CN.ITOD	integer	**CN.ITOBB**
	$F4 CN.ITOBB	binary (byte)	**CN.ITOBL**
	$F6 CN.ITOBW	binary (word)	**CN.ITOBW**
	$F8 CN.ITOBL	binary (long)	**CN.ITOD**
	$FA CN.ITOHB	hex (byte)	**CN.ITOHB**
	$FC CN.ITOHW	hex (word)	**CN.ITOHL**
	$FE CN.ITOHL	hex (long)	**CN.ITOHW**

Call parameters		Return parameters	
		D0	???
D1		D1	???
D2		D2	???
D3		D3	???
A0	pointer to buffer	A0	pointer to buffer
A1	pointer to stack	A1	pointer to stack
A2		A2	???
A3		A3	???

All addresses passed to these routines must be relative to A6.

These routines convert a value on the stack (pointed to by A1) to a set of ASCII characters in a buffer (pointed to by A0).

CN.HTOIB See the entry for **CN.DTOF** for details.

CN.HTOIL See the entry for **CN.DTOF** for details.

CN.HTOIW See the entry for **CN.DTOF** for details.

CN.ITOBB See the entry for **CN.FTOD** for details.

CN.ITOBL See the entry for **CN.FTOD** for details.

CN.ITOBW See the entry for **CN.FTOD** for details.

CN.ITOD See the entry for **CN.FTOD** for details.

CN.ITOHB See the entry for **CN.FTOD** for details.

CN.ITOHL See the entry for **CN.FTOD** for details.

CN.ITOHW See the entry for **CN.FTOD** for details.

Vector $122

IO.NAME

Decode a device name

Call parameters		Return parameters	
D1		D1	???
D2		D2	???
D3		D3	???
A0	pointer to name	A0	preserved
A1		A1	???
A2		A2	???
A3	pointer to parameters	A3	preserved

Error returns:

ERR.NF not recognised
ERR.BN name recognised – but bad parameters

This routine parses a device name. Given a device name and a description of the syntax of the name to be checked against and for the possible parameters to be appended to it, the routine determines whether the name is recognised, and extracts the parameters if it is. The device name is formed using four components:

Name	ASCII characters, normally letters. Case is ignored.
Separator	Single ASCII character. Case is ignored.
Number	Decimal number in the range 0 to 32767
Code	One of a list of ASCII characters

On entry to the routine, A0 must point to the device name (which is in the usual Qdos string format), A3 must point to an area of memory which is sufficient to hold the decoded parameter values, and A6 must point to the base of system variables. The device description starts 6 bytes after the call, and is in the following format:

word number of characters in the device name to be checked for
words the characters of the device name to be checked for
word number of parameters
For each parameter, one of the following options:
byte space, byte separator, word default value (numeric with separator)
word negative number, word default value (numeric with no separator)
word positive number of possible codes, bytes for the ASCII codes

Note that all letters must be in upper case.

For each numeric parameter value in the description, the utility will return either the value given in the device name, or the default. For each list of codes 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',180  window size
DC.W  'A',32,'X',16    window position
DC.W  '_',128          keyboard queue length
```

Device name	Parameters
CON	448,180,0,0,128
CON_256	**256**,180,0,0,128
con__60	448,180,0,0,**60**
cona0x12	448,180,**0,12,128**
con_256x64a64x128_20	**256,64,64,128,20**

The SER description is:

DC.W	3,'SER'	RS232 serial device
DC.W	4	four parameters
DC.W	−1,1	port number (default 1)
DC.W	4,'OEMS'	parity (odd/even/mark/space)
DC.W	2,'IH'	ignore/use handshaking
DC.W	3,'RZC'	Raw/use CTRLZ/use CR

Device name	Parameters
SER	1,0,0,0
SERE	1,1,0,0
ser2miZ	2,3,1,2

If the name is not matched, the routine returns immediately after the call with **ERR.NF** in D0. If the name is matched but the additional information is incorrect, it returns 2 bytes after the call with **ERR.BN** in D0. If a match is found, it returns 4 bytes after the call with D0=0.

IO.QEOF See the entry for **IO.QSET** for details.

IO.QIN See the entry for **IO.QSET** for details.

IO.QOUT See the entry for **IO.QSET** for details.

IO.QSET

Vector			
	$DC IO.QSET	set up a queue	**IO.QSET**
	$DE IO.QTEST	test status of queue	**IO.QEOF**
	$E0 IO.QIN	put byte into queue	**IO.QIN**
	$E2 IO.QOUT	extract byte from queue	**IO.QUOT**
	$E4 IO.QEOF	put end of file marker into queue	**IO.QTEST**

Call parameters		Return parameters	
D1.L	queue length or data	D1	data
D2		D2	preserved/free space
D3		D3	preserved
A0		A0	preserved
A1		A1	preserved
A2	pointer to queue	A2	preserved
A3		A3	modified by QIN, QOUT, QTEST, QSET

Error returns:

ERR.NC queue is full (QIN) or empty (QOUT, QTEST)
ERR.EF end of file reached (QOUT, QTEST)

The data length should be less than 32767

IO.QTEST See the entry for **IO.QSET** for details.

IO.SERIO See the entry for **IO.SERQ** for details.

Vector $E8 IO.SERQ direct queue handling
$EA IO.SERIO general IO handling

IO.SERQ
IO.SERIO

Call parameters		Return parameters	
D1	standard IOSS value	D1	standard IOSS value
D2	standard IOSS value	D2	standard IOSS value
D3	standard IOSS value	D3	???
A0	standard IOSS value	A0	preserved
A1	standard IOSS value	A1	standard IOSS value
A2		A2	???
A3		A3	???

Error returns:

ERR.BP undefined action
or errors returned from supplied routines

These routines must be called from supervisor mode, with A6 pointing to the base of system variables. It may not be called from a task which services an interrupt.

IO.SERQ is a direct queue handling routine. When the channel definition block is set up for simple serial I/O then the 7th and 8th long words 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 zero.

IO.SERQ should be called with standard IOSS values in D0, D1, D2, D3, A0 and A1.

For serial I/O where the operations for byte input and output are not 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

testing for pending input, (next byte in D1)
fetch byte, (byte in D1)
send byte. (byte in D1)

The use of absolute addresses for these may prove awkward; so the entry to this routine is best included in the physical definition block for the driver:

```
at $28(A3) or similar          or

387800E8   MOVE.W  $E8,A4                  DC.L    TEST
4E94       JSR     (A4)                    DC.L    FETCH
           DC.L    TEST                    DC.L    SEND
           DC.L    FETCH          4E75     RTS
           DC.L    SEND
4E75       RTS

invoked by                     or

           JSR     $28(A3)                 PEA     $28(A3)
                                           MOVE.W  $E8,A4
                                           JMP     (A4)
```

For the calls to the three service routines D0 should be returned as the error code, D1 to D3 and A1 to A3 inclusive are volatile.

Both of these calls treat actions 0, 1, 2, 3, 5 and 7, the header set and read actions and load and save: for undefined actions they return **ERR.BP**.

Vector			**MD.READ**
	$124 MD.READ	read a sector	**MD.SECTR**
	$126 MD.WRITE	write a sector	**MD.VERIN**
	$128 MD.VERIN	verify a sector	**MD.WRITE**
	$12A MD.SECTR	read a sector header	

Call parameters		Return parameters	
D1		D1	file nr (read/verify)
D2		D2	block nr (read/verify)
D7		D7	sector nr (read headr)
A0		A0	???
A1	pointer to start of bufr	A1	pointer to end of bufr
A2		A2	???
A3	$18020	A3	$18020

Error returns:

MD.WRITE	none
MD.READ, MD.VERIN	normal – failed return+2 OK
MD.SECTR	normal – bad medium return+2 – bad sector header return+4 – OK

The microdrive support routines are vectored to simplify the writing of file recovery programs. On entry A3 must point to the microdrive control register, and the interrupts must be disabled. All registers except A3 and A6 are treated as volatile.

These routines do not set D0 on return but have multiple returns.

Before calling **MD.WRITE** the stack pointer must point to a word: the file number and the block number of the sector to be written are in the high and low byte respectively.

These vectors point to $4000 bytes before the actual entry point. The following code may be used:

```
MOVE.W aa.aaaa,An
JSR $4000(An)
```

MD.SECTR See the entry for **MD.READ** for details.

MD.VERIN See the entry for **MD.READ** for details.

MD.WRITE See the entry for **MD.READ** for details.

Vector $C0

MM.ALCHP

Allocate common heap area

Call parameters	Return parameters
D1.L space required	D1.L space allocated
D2	D2 ???
D3	D3 ???
A0	A0 base of area allocated
A1	A1 ???
A2	A2 ???
A3	A3 ???

Error returns:

OM out of memory

This routine must be called from supervisor mode, with A6 pointing to the base of system variables. It may not be called from a task which services an interrupt.

The space requested must include room for the heap entry header. For simple heap entries this is 16 bytes long, for IOSS channels this is 24 bytes long.

The address of the heap area is the base of the area allocated, not the base of the area which may be used (contrast with trap #1 D0=$ 18 and $19).

The area allocated is cleared to zero.

Vector $D8

MM.ALLOC

Allocate an area in a heap

Call parameters		Return parameters	
D1.L	length required	D1.L	length allocated
D2		D2	???
D3		D3	???
A0	ptr to ptr to free space	A0	base of area allocated
A1		A1	???
A2		A2	???
A3		A3	???

Error returns:

OM no free space large enough

See section 4.1 for details of the heap allocation mechanism.

Vector $DA

MM.LNKFR

Link a free space (back) into a heap

Call parameters		Return parameters	
D1.L	length to link in	D1	???
D2		D2	???
D3		D3	???
A0	base of new space	A0	???
A1	ptr to ptr to free space	A1	???
A2		A2	???
A3		A3	???

Vector $C2

MM.RECHP

Release common heap space

Call parameters		Return parameters	
D1		D1	???
D2		D2	???
D3		D3	???
A0	base of area to release	A0	???
A1		A1	???
A2		A2	???
A3		A3	???

This routine must be called from supervisor mode, with A6 pointing to the base of system variables. It may not be called from a task which services an interrupt. See entry for **MM.ALCHP**

Vector $11C RI.EXEC executes an operation
$11E RI.EXECB executes a list of operations

RI.EXEC
RI.EXECB

Call parameters		Return parameters	
D0.W	operation (RI.EXEC)	D0	error code
D1		D1	preserved
D2		D2	preserved
D3		D3	preserved
A0		A0	preserved
A1	pointer to arith stack	A1	updated
A2		A2	preserved
A3	ptr to operation list	A3	preserved
A4	ptr to base of var area	A4	preserved

Error returns:

OV arithmetic overflow

All addresses passed to these routines must be relative to A6.

The arithmetic package is available for general use through two vectors: the first executes a single operation; the second executes a list of operations.

The package operates on floating point numbers on a downward stack pointed to by (A6,A1.L). It operates on the top of the stack (TOS) which is pointed to by (A6,A1.L), and the next on stack (NOS) at 6(A6,A1.L).

See section 9.5 for details of the floating point format.

The interpreter takes two types of operation codes. The first is a true arithmetic operation with an operation code between $02 and $30 inclusive, the second is a negative code between $FFFF and $FF31 inclusive: this indicates a load or store operation of a floating point number to or from the location given by the calculation (A6.L+A4.L+opcode$FFFE). If bit 0 of the opcode is clear the operation is a load (A1 decremented by 6, creating a new TOS), if it is set the operation is a store (A1 incremented by 6, NOS−−›TOS)

For **RI.EXEC** the operation code should be passed as a word. For **RI.EXECB** the operation codes are in a table of bytes pointed to by A3. The table is terminated by a zero byte.

Note: for the function EXP, D7 should be set to zero or an erroneous value will be returned.

The operation codes for the interpreter are as follows:

CODE		function change to A1	
$02	**RI.NINT**	nearest integer to TOS +4	
$04	**RI.INT**	truncate TOS to integer +4	
$06	**RI.NLINT**	nearest long integer to TOS +2	
$08	**RI.FLOAT**	integer TOS to floating point −4	
$0A	**RI.ADD**	add TOS to NOS +6	
$0C	**RI.SUB**	subtract TOS from NOS +6	
$0E	**RI.MULT**	multiply TOS by NOS +6	
$10	**RI.DIV**	divide TOS into NOS +6	
$12	**RI.ABS**	positive value of TOS 0	
$14	**RI.NEG**	negate TOS 0	
$16	**RI.DUP**	duplicate TOS −6	
$18	**RI.COS**	cosine	)
$1A	**RI.SIN**	sine	)
$1C	**RI.TAN**	tangent	)
$1E	**RI.COT**	cotangent	)
$20	**RI.ASIN**	arcsine	)
$22	**RI.ACOS**	arccosine	) change TOS only
$24	**RI.ATAN**	arctangent	) A1 unchanged
$26	**RI.ACOT**	arccotangent	)
$28	**RI.SQRT**	square root	)
$2A	**RI.LN**	natural logarithm	)
$2C	**RI.LOG10**	logarithm to base 10	)
$2E	**RI.EXP**	exponential	)
$30	**RI.POWFP**	NOS to the power of TOS +6	

UT.COM See the entry for **UT.WINDW** for details.

Vector $E6

UT.CSTR

Compare two strings

Call parameters		Return parameters	
D0.B	comparison type	D0.L	−1, 0 or +1
D1		D1	preserved
D2		D2	preserved
D3		D3	preserved
A0	base of string 0 wrt A6	A0	preserved
A1	base of string 1 wrt A6	A1	preserved
A2		A2	preserved
A3		A3	preserved
A6	base address register	A6	preserved

All addresses passed to this routine must be relative to A6.

D0 (and the status register) is set negative if the string at (A6,A0) is less than the string at (A6,A1) etc.

UT.ERRO
UT.ERR

Vector $CA UT.ERR0	write error message to channel 0
$CC UT.ERR	write error message to given channel

Call parameters		Return parameters	
D0.l	error code	D0.l	preserved
D1		D1	preserved
D2		D2	preserved
D3		D3	preserved
A0	channel ID (UT.ERR only)	A0	preserved
A1		A1	preserved
A2		A2	preserved
A3		A3	preserved

These routines must be called from user mode.

These routines exist for writing simple messages to a channel. They are basic error message handlers which write a standard or device driver supplied error message to either the command channel 0, or else to a defined channel.

UT.LINK
UT.UNLINK

Vector $D2 UT.LINK		link an item into a list
$D4 UT.UNLNK		unlink an item from a list
Call parameters		**Return parameters**
D1		D1 preserved
D2		D2 preserved
D3		D3 preserved
A0	base of item (un)linked	A0 preserved
A1	pointer to previous item	A1 updated
A2		A2 preserved
A3		A3 preserved

These two routines are provided for handling linked lists.

These routines are passed the base address of the item to be linked or unlinked, and a pointer which points to either the pointer to the first item in the list, or to an item in the list.

When an item is linked in, it will be linked in at the start of the list, or, if the pointer was to an item in the list, after that item.

When an item is removed, the pointer may point to the pointer to the first item in the list, or to any item in the list before the item to be removed.

When starting a new list, the pointer to the first item in the list must be zero.

Each item in the list must have 4 bytes reserved at the start for the link pointer.

Vector $CE

UT.MINT

Convert an integer to ASCII and send it to the defined channel

Call parameters		Return parameters	
D1.W	integer value	D1	???
D2		D2	???
D3		D3	???
A0	channel ID	A0	preserved
A1		A1	???
A2		A2	preserved
A3		A3	preserved

Error returns:

All the usual IO

This routine ought usually to be called from user mode.

Vector $D0

UT.MTEXT

Send a message to a channel

Call parameters		Return parameters	
D1		D1	???
D2		D2	???
D3		D3	???
A0	channel ID	A0	preserved
A1	base of message	A1	???
A2		A2	preserved
A3		A3	preserved

Error returns:

All the usual IO

This routine ought usually to be called from user mode.

The above routines (**UT.MINT** and **UT.MTEXT**) are provided to write parts of more complex messages to a defined channel.

The message is in the form of a text string: number of characters (word) followed by the characters in ASCII. If a new line is required at the end of the message, this should be included in the message. If the channel is 0 then D3 will be returned 0, otherwise D3 will be returned −1. In version V1.03 and earlier, D0 is set to the error return but is not tested so the condition codes will not be correct. As a special concession, interrupt servers and other supervisor mode routines can call these routines with A0=0. If the command channel is in use, they will attempt to use channel 1. This operation is not recommended, but it does seem to work!

UT.SCR See the entry for **UT.WINDW** for details.

UT.UNLNK See the entry for **UT.LINK** for details.

UT.WINDW

Vector $C4	UT.WINDW	set up a window using a supplied name	**UT.CON** **UT.SCR**
	$C6 UT.CON	set up console window	
	$C8 UT.SCR	set up screen window	

Call parameters		Return parameters	
D1		D1	???
D2		D2	???
D3		D3	???
A0	ptr to name (WINDW only)	A0	channel ID
A1	ptr to parameter block	A1	???
A2		A2	???
A3		A3	???

Error returns:

BN bad device name (WINDW only)
OM out of memory
NO out of channels
OR window is off-screen

The above three routines, which must be called in user mode, set up console or screen windows using a parameter list which follows the call statement. In the first case, the window is opened using a name which has been supplied, a block of parameters defining the border, and the paper, strip and ink colours. The window is set up and cleared for use.

The second and third routines define the window using an additional block of four words.

The parameter block is as follows:

$00	border colour (byte)		
$01	border width (byte)		
$02	paper/strip colour (byte)		
$03	ink colour (byte)		
$04	width (word)	)	
$06	height (word)	)	not required for **UT.WINDW**
$08	X origin (word)	)	
$0A	Y origin (word)	)	

17.0 Qdos System Standards

In order to make best use of the third-party work, both software and hardware, currently going on on the QL, a number of Sinclair standards have been produced.

1. Floppy disc standard – This covers the physical layout, formatting, directory structure and disk handling of floppy disks under Qdos.

2. Relocatable object file standard – To allow the linking of separately compiled modules, potentially written in different languages.

These standards are available on application to Sinclair Research at the address in the introduction.

18.0 Qdos Keys

The following sections contain keys for various features of Qdos. These keys provide a definition for several of the data structures within Qdos.

18.1 Error Keys

The following keys indicate error messages already defined in the system. A positive error code is taken as an address of a user-supplied error message. See the **Concepts** manual for a fuller description of the way in which these are used by the procedures built into SuperBASIC.

ERR.NC	−1	operation not complete
ERR.NJ	−2	not a valid job
ERR.OM	−3	out of memory
ERR.OR	−4	out of range
ERR.BO	−5	buffer overflow
ERR.NO	−6	channel not open
ERR.NF	−7	file or device not found
ERR.EX	−8	file already exists
ERR.IU	−9	file or device in use
ERR.EF	−10	end of file
ERR.DF	−11	drive full
ERR.BN	−12	bad device name
ERR.TE	−13	transmission error
ERR.FF	−14	format failed
ERR.BP	−15	bad parameter
ERR.FE	−16	file error
ERR.XP	−17	error in expression
ERR.OV	−18	arithmetic overflow
ERR.NI	−19	not implemented (yet)
ERR.RO	−20	read only
ERR.BL	−21	bad line (syntax error in BASIC)

18.2 System Variables

The following list gives the offset of each system variable from the base of the system variables (whose position can be found using the **MT.INF** trap), together with the length of the variable.

SV_IDENT	$00	word	identification word

The following variables are the pointers which define the current state of the Qdos memory map.

SV_CHEAP	$04	long	base of common heap area
SV_CHPFR	$08	long	first free space in common heap area
SV_FREE	$0C	long	base of free area
SV_BASIC	$10	long	base of basic area
SV_TRNSP	$14	long	base of transient program area
SV_TRNFR	$18	long	first free space in transient program area
SV_RESPR	$1C	long	base of resident procedure area
SV_RAMT	$20	long	top of ram (+1)
SV_RAND	$2E	word	random number
SV_POLLM	$30	word	count of poll interupts missed
SV_TVMOD	$32	byte	0 if not TV display
SV_SCRST	$33	byte	screen status (0 = active)
SV_MCSTA	$34	byte	current value of display control register
SV_PCINT	$35	byte	current value of interrupt control/mask register
SV_NETNR	$37	byte	network station number

The following system variables are pointers to the list of tasks and drivers.

SV_I2LST	$38	long	pointer to list of interrupt 2 drivers
SV_PLIST	$3C	long	pointer to list of polled tasks
SV_SHLST	$40	long	pointer to list of scheduler tasks
SV_DRLST	$44	long	pointer to list of device drivers
SV_DDLST	$48	long	pointer to list of directory device drivers
SV_KEYQ	$4C	long	pointer to a keyboard queue
SV_TRAPV	$50	long	pointer to the trap redirection table

The following system variables are pointers to the resource management tables. The slave block tables have 8 byte entries, whilst the others have 4 byte entries.

SV_BTPNT	$54	long	pointer to most recent slave block entry
SV_BTBAS	$58	long	pointer to base of slave block table
SV_BTTOP	$5C	long	pointer to top of slave block table
SV_JBTAG	$60	word	current value of job tag
SV_JBMAX	$62	word	highest current job number
SV_JBPNT	$64	long	pointer to current job table entry
SV_JBBAS	$68	long	pointer to base of job table
SV_JBTOP	$6C	long	pointer to top of job table
SV_CHTAG	$70	word	current value of channel tag
SV_CHMAX	$72	word	highest current channel number
SV_CHPNT	$74	long	pointer to last channel checked
SV_CHBAS	$78	long	pointer to base of channel table
SV_CHTOP	$7C	long	pointer to top of channel table

The following variables contain information about how to treat the keyboard, and about other aspects of the IPC and serial port communications. **SV_CAPS**, **SV_ARDEL**, **SV_ARFRQ** and **SV_CSUB** can safely be poked.

SV_CAPS	$88	word	caps lock
SV_ARBUF	$8A	word	autorepeat buffer
SV_ARDEL	$8C	word	autorepeat delay
SV_ARFRQ	$8E	word	autorepeat 1/freq
SV_ARCNT	$90	word	autorepeat count
SV_CQCH	$92	word	keyboard change queue character code
SV_SOUND	$96	word	sound status
SV_SER1C	$98	long	receive channel 1 queue address
SV_SER2C	$9C	long	receive channel 2 queue address
SV_TMODE	$A0	byte	ZX8302 transmit mode (includes baudrate)
SV_CSUB	$A2	long	subroutine to jump to on CAPSLOCK
SV_TIMO	$A6	word	timeout for switching transmit mode
SV_TIMOV	$A8	word	value of switching timeout (two characters)
SV_ FSTAT	$AA	word	flashing cursor status

SV_MDRUN	$EE	byte	which drive is running?
SV_MDCNT	$EF	byte	microdrive run-up run-down counter
SV_MDDID	$F0	8 bytes	drive ID*4 of each microdrive
SV_MDSTA	$F8	8 bytes	status 0= no pending ops
SV_FSDEF	$100	16*long	pointers to file system physical definition
SV_FSLST	$140	long	pointer to list of file channel definitions

The following area, between $180 and $480 is reserved for the supervisor stack. There is no explicit stack protection in the code, although the stack should be of sufficient size for most normal purposes.

18.3 SuperBASIC Variables

Note that the system variable **SV_BASIC** points to the bottom of the SuperBASIC area, where its job header, which is $68 bytes long, is located. The value of A6 used during the interpreter points to the address immediately above the job header, which contains a set of variables formatted as shown in this table.

The first part of the area holds the pointers to the various areas of memory used by the interpreter: it defines the partitioning of SuperBASIC's own area of memory.

BV_START	0		start of pointers
BV_BFBAS	$00	long	buffer base
BV_BFP	$04	long	buffer running pointer
BV_TKBAS	$08	long	token list
BV_TKP	$0C	long	
BV_PFBAS	$10	long	program file
BV_PFP	$14	long	
BV_NTBAS	$18	long	name table
BV_NTP	$1C	long	
BV_NLBAS	$20	long	name list
BV_NLP	$24	long	
BV_VVBAS	$28	long	variable values
BV_VVP	$2C	long	
BV_CHBAS	$30	long	channel table
BV_CHP	$34	long	
BV_RTBAS	$38	long	return table

BV_RTP	$3C	long	
BV_LNBAS	$40	long	line number table
BV_LNP	$44	long	
BV_CHANGE	$48		change of direction marker
BV_BTP	$48	long	backtrack stack during parsing
BV_BTBAS	$4C	long	
BV_TGP	$50	long	temporary graph stack during parsing
BV_TGBAS	$54	long	
BV_RIP	$58	long	arithmetic stack
BV_RIBAS	$5C	long	
BV_SSP	$60	long	system stack (real one!)
BV_SSBAS	$64	long	
BV_ENDPT	$64		end of pointers
BV_LINUM	$68	word	current line number
BV_LENGTH	$6A	word	current length
BV_STMNT	$6C	byte	current statement on line
BV_CONT	$6D	byte	continue ($80) or stop (0) processing
BV_INLIN	$6E	byte	processing in-line clause or not loop (1), other ($FF), not (0)
BV_SING	$6F	byte	single line execution on ($FF) or off (0)
BV_INDEX	$70	word	name table row of last in-line loop index read
BV_VVFREE	$72	long	first free space in variable value table
BV_SSSAV	$76	long	saved sp for out/mem to go back to
BV_RAND	$80	long	random number
BV_COMCH	$84	long	command channel
BV_NXLIN	$88	word	which line number to start after
BV_NXSTM	$8A	byte	which statement to start after
BV_COMLN	$8B	byte	command line saved ($FF) or not (0)
BV_STOPN	$8C	word	which stop number set
BV_EDIT	$8E	byte	program has been edited ($FF) or not (0)
BV_BRK	$8F	byte	there has been a break (0) or not ($80)
BV_UNRVL	$90	byte	need to unravel ($FF) or not (0)
BV_CNSTM	$91	byte	statement to CONTINUE from
BV_CNLNO	$92	word	line to CONTINUE from

BV_DALNO	$94	word	current DATA line number
BV_DASTM	$96	byte	current DATA statement number
BV_DAITM	$97	byte	next DATA item to read
BV_CNIND	$98	word	in-line loop index to CONTINUE with
BV_CNINL	$9A	byte	in-line loop flag for CONTINUE
BV_LSANY	$9B	byte	whether checking list ($FF) or not (0)
BV_LSBEF	$9C	word	invisible top line
BV_LSBAS	$9E	word	bottom line in window
BV_LSAFT	$A0	word	invisible bottom line
BV_LENLN	$A2	word	length of window line
BV_MAXLN	$A4	word	max nr of window lines The 2 words immediately following this will be overwritten on changing lenln and maxln
BV_TOTLN	$A6	word	nr of window lines so far
BV_AUTO	$AA	byte	whether AUTO/EDIT on ($FF) or off (0)
BV_PRINT	$AB	byte	print from prtok ($FF) or leave in buffer (0)
BV_EDLIN	$AC	word	line number to edit next
BV_EDINC	$AE	word	increment on edit range
BV_TKPOS	$B0	long	pos of A4 in tklist on entry to PROC
BV_PTEMP	$B4	long	temp pointer for GO_PROC
BV_UNDO	$B8	byte	undo rt stack IMMEDIATELY then redo procedure
BV_ARROW	$B9	byte	down ($FF) or up (01) or no (00) arrow
BV_LSFIL	$BA	word	fill window when relisting at least to here
BV_WRLNO	$BC	word	when error line number
BV_WRSTM	$BE	byte	when error statement
BV_WRINL	$BF	byte	when error in-line ($FF) or not (0)
BV_WHERR	$C0	byte	processing when error($80) or not (0)
BV_ERROR	$C2	long	last error code
BV_ERLIN	$C6	word	line number of last error
BV_WVNUM	$C8	word	number of watched (WHEN) variables
BV_WVBAS	$CA	long	base of WHEN variable table wrt VVBAS
BV_END	$100		top of BV area

18.4 Offsets on BASIC Channel Definitions

The following section gives the format of an entry in the SuperBASIC channel table. These entries can be monitored or modified by user-defined SuperBASIC procedures which need to have a channel attached using a '#n' construct.

CH.ID	$00		channel id
CH.CCPY	$04	float	current cursor position, y
CH.CCPX	$0A	float	current cursor position, x
CH.ANGLE	$10	float	turtle angle
CH.PEN	$16	byte	pen status (up or down)
CH.CHPOS	$20	word	character position on line
CH.WIDTH	$22	word	width of line in characters
CH.SPARE	$24		..spare..
CH.LENCH	$28		length of a channel definition block

18.5 Job Header and Save Area Definitions

The location of the job table can be found by looking at the system variables **SV_JBBAS** and **SV_JBTOP**. Each entry in the table is a longword pointing to a block of $68 bytes in the format given here.

JB_LEN*	$00	long	total length of job area
JB_START	$04	long	start address on activation (usually 0)
JB_OWNER	$08	long	job ID of the owner of this job
JB_HOLD	$0C	long	ptr to byte to be cleared when job released
JB_TAG*	$10	word	tag for this job, allocated by MT.CJOB
JB_PRIOR	$12	byte	current accumulated priority: set to zero when the job is executing, incremented on each scheduler call if the job is active but not executing
JB_PRINC	$13	byte	priority increment (the actual priority of the job) set to zero if the job is inactive SuperBASIC activates jobs at priority $20
JB_STAT*	$14	word	job status 0 => not suspended >0 => number of frame times to release −1 => suspended −2 => waiting for another job to finish

JB_RELA6	$16	byte	MSB set if next trap #2 or #3 has addressing relative to A6
JB_WFLAG	$17	byte	set if there is a job waiting on completion of this one
JB_WJOB	$18	long	job ID of waiting job
JB_TRAPV	$1C	long	pointer to trap redirection vectors
JB_D0	$20		save offset of D0
JB_D1	$24		save offset of D1
JB_D2	$28		save offset of D2
JB_D3	$2C		save offset of D3
JB_D4	$30		save offset of D4
JB_D5	$34		save offset of D5
JB_D6	$38		save offset of D6
JB_D7	$3C		save offset of D7
JB_A0	$40		save offset of A0
JB_A1	$44		save offset of A1
JB_A2	$48		save offset of A2
JB_A3	$4C		save offset of A3
JB_A4	$50		save offset of A4
JB_A5	$54		save offset of A5
JB_A6	$58		save offset of A6
JB_A7	$5C		save offset of A7
JB_USP	$5C		save offset of USP
JB_SR	$60		save offset of SR
JB_PC	$62		save offset of PC
JB_END	$68		

Thus the job identified by ‹job-ID› starts at ((SV_JBBAS)+4*‹job-ID›.W), and the most significant word of ‹job-ID› must match the tag held at 10H on from this address (otherwise that job no longer exists). A negative ‹job-ID› implies that the job no longer exists, as does a value of ‹job-ID›.W which is greater than the length of the job table held in **SV_JBMAX**.

Entries marked by * should not be modified. Other entries may be modified by a trap, or may be changed directly with caution.

18.6 Memory Block Table Definitions

The following keys define the format of the start of a slave block.

BT_STAT	$00	byte	drive ID/status byte – see below
BT_PRIOR	$01	byte	block priority
BT_SECTR	$02	word	sector number (Microdrive*2)
BT_FILNR	$04	word	file number (Microdrive) logical
BT_BLOCK	$06	word	block number (Microdrive) location
BT_END	$08		

The most significant 4 bits of the status byte contain the pointer to the physical device block **SV_FSDEF**, the least significant are the status codes:

BT.UNAV	00000000B	block is unavailable to file system
BT.EMPTY	00000001B	block is empty
BT.RREQ	00001001B	block required to be read from microdrive
BT.TRUE	00000011B	block is a true representation of file
BT.AVER	00001011B	block is awaiting verify
BT.UPDT	00000111B	block is updated

Status code masks:

BT.ACTN	00001100B	check for read or write request
BT.INUSE	00001110B	check if a file block in use

Bits of status codes:

BT..FILE	0	1 if a file block
BT..ACCS	1	1 if contents may be accessed
BT..WREQ	2	1 if block required to be written
BT..RDVR	3	1 if block required to be read/verified

18.7 Channel Definitions

The position of a channel definition block corresponding to a given channel ID can be found using a similar method to that used for finding the block for a job described in section 3.1. The relevant system variables are **SV_CHBAS** and **SV_CHMAX**.

Channel definition header for all channels:

CH_LEN	$00	long	length of definition block
CH_DRIVR	$04	long	address of driver
CH_OWNER	$08	long	owner job
CH_RFLAG	$0C	long	address to be set when space released
CH_TAG	$10	word	channel tag
CH_STAT	$12	byte	status – 0 OK, negative waiting –1 A1 abs, $80 A1 rel A6
CH_ACTN	$13	byte	stored action for waiting job
CH_JOBWT	$14	long	ID of job waiting on IO
CH_END	$18		

Extended channel definition for plain serial queues:

CH_QIN	$18	long	pointer to input queue (or zero)
CH_QOUT	$1C	long	pointer to output queue (or zero)
CH_QEND	$20		

Device driver header:

CH_NEXT	$00	long	pointer to next driver
CH_INOUT	$04	long	entry for input and output
CH_OPEN	$08	long	entry for open
CH_CLOSE	$0C	long	entry for close

The following are for directory devices (file system) only:

CH_SLAVE	$10	long	entry for slaving blocks
CH_RENAM	$14	long	entry reserved for rename
CH_FORMT	$1C	long	entry for format medium
CH_DFLEN	$20	long	length of physical definition block
CH_DRNAM	$24	2+n bytes	drive name

18.8 File System Definition Blocks

File system channel definition block format:

FS_NEXT	$18	long	link to next file system channel
FS_ACCES	$1C	byte	access mode
FS_DRIVE	$1D	byte	drive ID
FS_FILNR	$1E	word	file number
FS_NBLOK	$20	word	block containing next byte
FS_NBYTE	$22	word	next byte in block
FS_EBLOK	$24	word	end of file (block)
FS_EBYTE	$26	word	end of file (byte in block)
FS_CBLOK	$28	long	pointer to table for current slave block
FS_UPDT	$2C	byte	set if file is updated
FS_FNAME	$32	2+36	file name
FS_SPARE	$58	72 bytes	
FS_END	$A0		

The common part of a physical definition block

FS.NMLEN	$24		max length of file name
FS.HDLEN	$40		length of file system header
FS_DRIVR	$10	long	pointer to driver
FS_DRIVN	$14	byte	drive number
FS_MNAME	$16	word+10	bytes medium name
FS_FILES	$22	byte	number of files open

18.9 Screen Driver Data Block Definition

This is the format of the block handed to a screen driver operation.

SD_XMIN	$18	word	window top LHS
SD_YMIN	$1A	word	
SD_XSIZE	$1C	word	window size
SD_YSIZE	$1E	word	
SD_BORWD	$20	word	border width
SD_XPOS	$22	word	cursor position
SD_YPOS	$24	word	
SD_XINC	$26	word	cursor increment
SD_YINC	$28	word	
SD_FONT	$2A	2*long	font addresses
SD_SCRB	$32	long	base address of screen
SD_PMASK	$36	long	paper colour mask
SD_SMASK	$3A	long	strip colour mask
SD_IMASK	$3E	long	ink colour mask
SD_CATTR	$42	byte	character attributes
SD_CURF	$43	byte	cursor flag 0=suppressed, ›0=visible
SD_PCOLR	$44	byte	paper colour byte
SD_SCOLR	$45	byte	strip colour byte
SD_ICOLR	$46	byte	ink colour byte
SD_BCOLR	$47	byte	border colour byte
SD_NLSTA	$48	byte	new line status (›0 implicit, ‹0 explicit)
SD_FMOD	$49	byte	fill mode (0=off, 1=on)
SD_YORG	$4A	float	graphics window y-origin
SD_XORG	$50	float	graphics window x-origin
SD_SCAL	$56	float	graphics scale factor
SD_FBUF	$5C	long	pointer to fill buffer
SD_FUSE	$60	long	pointer to user defined fill vectors
SD_LINEL	$64	word	line length in bytes
SD_END	**$68**		

18.10 Queue Header Definitions

The following is the format of the header of a queue manipulated using the system's built-in queue handling routines.

Q_EOFF	$00	bit	end of file flag (MSbit)
Q_NEXTQ	$00	long	link to next queue
Q_END	$04	long	pointer to end of queue
Q_NEXTIN	$08	long	pointer to next location to put byte in
Q_NXTOUT	$0C	long	pointer to next location to take byte from
Q_QUEUE	$10		start of queue

18.11 Arithmetic Interpreter Operation Codes

The following are the codes for the operations which can be performed on the QL through the vectored routines which access the arithmetic interpreter.

RI.TERM	$00	terminator byte
RI.NINT	$02	nearest integer to top of stack (tos)
RI.INT	$04	truncate tos to integer
RI.NLINT	$06	nearest long integer to tos
RI.FLOAT	$08	integer tos to floating point
RI.ADD	$0A	add tos to next on stack (nos)
RI.SUB	$0C	subtract tos from nos
RI.MULT	$0E	multiply tos by nos
RI.DIV	$10	divide tos into nos
RI.ABS	$12	positive value of tos
RI.NEG	$14	negate tos
RI.DUP	$16	duplicate tos
RI.COS	$18	cosine
RI.SIN	$1A	sine
RI.TAN	$1C	tangent
RI.COT	$1E	cotangent
RI.ASIN	$20	arcsine
RI.ACOS	$22	arccosine
RI.ATAN	$24	arctangent
RI.ACOT	$26	arccotangent

RI.SQRT	$28	square root
RI.LN	$2A	natural log
RI.LOG10	$2C	logarithm to base 10
RI.EXP	$2E	exponential
RI.POWFP	$30	nos to power of tos
RI.MAXOP	$30	highest valid opcode
RI.LOAD	$00	load operation bit
RI.STORE	$01	store operation bit

18.12 IPC Link Commands

These can be used with the **MT.IPCOM** trap.

RSET_CMD	0	system reset
STAT_CMD	1	report input status
OPS1_CMD	2	open RS232 channel 1
OPS2_CMD	3	open RS232 channel 2
CLS1_CMD	4	close RS232 channel 1
CLS2_CMD	5	close RS232 channel 2
RDS1_CMD	6	read RS232 channel 1
RDS2_CMD	7	read RS232 channel 2
RDKB_CMD	8	read keyboard
KBDR_CMD	9	keyboard direct read
INSO_CMD	10	initiate sound process
KISO_CMD	11	kill sound process
MDRS_CMD	12	Microdrive reduced sensitivity
BAUD_CMD	13	change baud rate
RAND_CMD	14	random number generator
TEST_CMD	15	test

18.13 Hardware Keys

The following are the addresses of the registers within the QL hardware.

PC_CLOCK	$18000	real time clock in seconds (long word)
PC_TCTRL	$18002	transmit control
PC_MCTRL	$18020	Microdrive control/status and IPC status
PC_IPCRD	$18020	IPC read is the same
PC_IPCWR	$18003	IPC write
PC_INTR	$18021	interrupt control/status

PC_TDATA	$18022	transmit data
PC_TRAK1	$18022	Microdrive read track 1
PC_TRAK2	$18023	Microdrive read track 2
MC_STAT	$18063	display control

The following is the format of the interrupt register.

PC.INTRG	$01	gap interrupt
PC.INTRI	$02	interface interrupt
PC.INTRT	$04	transmit interrupt
PC.INTRF	$08	frame interrupt
PC.INTRE	$10	external interrupt
PC.MASKG	$20	gap mask
PC.MASKI	$40	interface mask
PC.MASKT	$80	transmit mask

The following is the format of the transmit control register.

PC..SERN	3	serial port number
PC..SERB	4	0=serial IO
PC..DIRO	7	direct output
PC.BMASK	00000111B	baud rate mask
PC.NOTMD	11100111B	all bits except mode control
PC.MDVMD	00010000B	Microdrive mode
PC.NETMD	00011000B	network mode

The following is the format of the Microdrive control/ systems register.

PC..SEL	0	Microdrive select bit
PC..SCLK	1	Microdrive select clock bit
PC..WRIT	2	Microdrive write
PC..ERAS	3	Microdrive erase
PC..TXFL	1	Microdrive Xmit buffer full
PC..RXRD	2	Microdrive read buffer ready
PC..GAP	3	gap
PC..DTR1	4	DTR on port 1
PC..CTS2	5	CTS on port 2

Write masks:

PC.READ	0010B	read (or idle) Microdrive
PC.SELEC	0011B	select bit set
PC.DESEL	0010B	select bit not set
PC.ERASE	1010B	erase on/write off
PC.WRITE	1110B	erase and write

The following is the format of the display control register.

MC..BLNK	1	bit 1 blanks display
MC..M256	3	bit 3 sets 256 mode
MC..SCRN	7	bit 7 sets screen base

18.14 Trap Keys

This section gives a summary of all of the Qdos traps, together with their access keys passed in D0. All keys are in hex.

18.14.1 Trap 1 Keys (Manager Traps) –

MT.INF	$00	get system information
MT.CJOB	$01	create a job
MT.JINF	$02	get information on job
MT.RJOB	$04	remove a job
MT.FRJOB	$05	force remove a job
MT.FREE	$06	find out how much free space there is
MT.TRAPV	$07	set pointer to trap redirection vectors
MT.SUSJB	$08	suspend a job
MT.RELJB	$09	release a job
MT.ACTIV	$0A	activate a job
MT.PRIOR	$0B	set a job priority
MT.ALLOC	$0C	allocate a bit of a heap
MT.LNKFR	$0D	release a bit of a heap
MT.ALRES	$0E	allocate resident procedure area
MT.RERES	$0F	release resident procedure area
MT.DMODE	$10	set display mode
MT.IPCOM	$11	send IPC command
MT.BAUD	$12	set baud rate
MT.RCLCK	$13	read clock
MT.SCLCK	$14	set clock
MT.ACLCK	$15	adjust clock

MT.ALBAS	$16	allocate BASIC area
MT.REBAS	$17	release BASIC area
MT.ALCHP	$18	allocate space in common heap
MT.RECHP	$19	release space in common heap
MT.LXINT	$1A	link in external interrupt handler
MT.RXINT	$1B	remove external interrupt handler
MT.LPOLL	$1C	link in polled task
MT.RPOLL	$1D	remove polled task
MT.LSCHD	$1E	link in scheduler task
MT.RSCHD	$1F	remove scheduler task
MT.LIOD	$20	link in IO driver
MT.RIOD	$21	remove IO driver
MT.LDD	$22	link in directory driver
MT.RDD	$23	remove directory driver

18.14.2 Trap 2 Keys (I/O Management Traps) –

IO.OPEN	$01	open channel
IO.CLOSE	$02	close channel
IO.FORMT	$03	format medium
IO.DELET	$04	delete file
IO.OPEN	D3	keys:
IO.OLD	0	open old (exclusive) file or device
IO.SHARE	1	open old (shared) file
IO.NEW	2	open new (exclusive) file
IO.OVERW	3	overwrite (or open new) file
IO.DIR	4	open directory

18.14.3 Trap 3 Keys (I/O Traps) –

IO.PEND	$00	check for pending input
IO.FBYTE	$01	fetch a byte
IO.FLINE	$02	fetch a line of bytes
IO.FSTRG	$03	fetch a string of bytes
IO.EDLIN	$04	edit a line
IO.SBYTE	$05	send a byte
IO.SSTRG	$07	send a string of bytes
SD.EXTOP	$09	external operation (A3)
SD.PXENQ	$0A	pixel based size/position enquiry
SD.CHENQ	$0B	character based size/position enquiry
SD.BORDR	$0C	define window border

SD.WDEF	$0D	define window
SD.CURE	$0E	enable cursor
SD.CURS	$0F	suppress cursor
SD.POS	$10	absolute position
SD.TAB	$11	tab (horizontal position)
SD.NL	$12	newline
SD.PCOL	$13	previous column
SD.NCOL	$14	next column
SD.PROW	$15	previous row
SD.NROW	$16	next row
SD.PIXP	$17	set pixel position
SD.SCROL	$18	scroll whole window
SD.SCRTP	$19	scroll top of window
SD.SCRBT	$1A	scroll bottom of window
SD.PAN	$1B	pan window
SD.PANLN	$1E	pan cursor line
SD.PANRT	$1F	pan RHS of cursor line
SD.CLEAR	$20	clear whole window
SD.CLRTP	$21	clear top of window
SD.CLRBT	$22	clear bottom of window
SD.CLRLN	$23	clear cursor line
SD.CLRRT	$24	clear to right of cursor
SD.FOUNT	$25	set fount addresses
SD.RECOL	$26	recolour a window
SD.SETPA	$27	set paper colour
SD.SETST	$28	set strip colour
SD.SETIN	$29	set ink colour
SD.SETFL	$2A	set flash on/off
SD.SETUL	$2B	set underline on/off
SD.SETMD	$2C	set write mode
SD.SETSZ	$2D	set character size
SD.FILL	$2E	fill block
SD.DONL	$2F	do pending newline
SD.POINT	$30	set point in window
SD.LINE	$31	draw line
SD.ARC	$32	draw arc
SD.ELIPS	$33	draw ellipse
SD.SCALE	$34	set graphics scale
SD.FLOOD	$35	set fill mode/vectors
SD.GCUR	$36	set text cursor using graphics coords
SD.ROP	$37	rasterop
SD.DOT	$38	points in pixel coords

SD.LIN	$39	lines in pixel coords
FS.CHECK	$40	check all pending operations
FS.FLUSH	$41	flush buffers
FS.POSAB	$42	position file pointer (absolute)
FS.POSRE	$43	position file pointer (relative)
FS.MDINF	$45	information about medium
FS.HEADS	$46	set file header
FS.HEADR	$47	read file header
FS.LOAD	$48	load file
FS.SAVE	$49	save file

18.15 List of Vectored Routines

The following is a list of the vectored routines, together with the addresses of their associated vectors. All keys are in hex.

BP.INIT	$110	add m/c procs/fns to BASIC
BP.LET	$120	assign tos to variable
BV.CHRIX	$11A	reserve space on RI stack
CA.GTINT	$112	get word parameters to RI stack
CA.GTFP	$114	get floating point numbers
CA.GTSTR	$116	get strings
CA.GTLIN	$118	get long integers
CN.BTOIB	$104	ASCII binary to byte
CN.BTOIL	$108	ASCII binary to long
CN.BTOIW	$106	ASCII binary to word
CN.DATE	$EC	get ASCII date and time
CN.DAY	$EE	get ASCII day of week
CN.DTOF	$100	ASCII to floating point
CN.DTOI	$102	ASCII to integer
CN.FTOD	$F0	floating point to ASCII
CN.HTOIB	$10A	ASCII hex to byte
CN.HTOIL	$10E	ASCII hex to long
CN.HTOIW	$10C	ASCII hex to word
CN.ITOBB	$F4	byte to ASCII binary
CN.ITOBL	$F8	long to ASCII binary
CN.ITOBW	$F6	word to ASCII binary
CN.ITOD	$F2	word integer to ASCII
CN.ITOHB	$FA	byte to ASCII hex

CN.ITOHL	$FE	long to ASCII hex
CN.ITOHW	$FC	word to ASCII hex
IO.NAME	$122	decode a device name
IO.QSET	$DC	set up a queue
IO.QTEST	$DE	test status of queue
IO.QIN	$E0	put byte into queue
IO.QOUT	$E2	extract byte from queue
IO.QEOF	$E4	put EOF marker into queue
IO.SERQ	$E8	direct queue handling
IO.SERIO	$EA	general IO handling

The MD routines are indexed by $4000.

MD.READ	$124	read a sector
MD.WRITE	$126	write a sector
MD.VERIN	$128	verify a sector
MD.SECTR	$12A	read a sector header
MM.ALCHP	$C0	allocate common heap space
MM.ALLOC	$D8	allocate an area in a heap
MM.LNKFR	$DA	link free space back into heap
MM.RECHP	$C2	release common heap space
RI.EXEC	$11C	execute an operation
RI.EXECB	$11E	execute a list of operations
UT.CON	$C6	set up console window
UT.CSTR	$E6	compare two strings
UT.ERR	$CC	write error message to channel
UT.ERR0	$CA	write error message to channel zero
UT.LINK	$D2	link an item into a list
UT.MINT	$CE	convert integer to ASCII, put on chan
UT.MTEXT	$D0	send message to channel
UT.SCR	$C8	set up screen window
UT.UNLNK	$D4	unlink an item from a list
UT.WINDW	$C4	set up window using supplied name

19.0 Doing Business with Sinclair

The purpose of this section is to encourage those thinking of developing commercial software for the QL to consider offering it to Sinclair Research for publishing, promotion and distribution. There are various options offered to software houses, with varying degrees of Sinclair involvement and support.

The first option is that of full publication and manufacture by Sinclair, whereby the new product is taken as a master with draft documentation, packaged in Sinclair packaging style and sold under the Sinclair logo in all the outlets stocking Sinclair hardware products. The software house is thereby released completely from the tasks of production, packaging, promotion, distribution and sale. For such a proposal to be financially viable, Sinclair has to obtain an exclusive licence for the product on Sinclair computers, and Sinclair will pay a percentage royalty on every unit sold. The software house remains free, of course, to develop the software for other computers, should it wish to do so.

The second option is for the software house to give Sinclair an exclusive licence to distribute the product in Sinclair packaging, but to sell the product to Sinclair as a fully packaged finished product to Sinclair specification. In this way the software house remains responsible for production and packaging, with Sinclair undertaking promotion, distribution and sale.

The third option is for the software house to retain responsibility for production, packaging, promotion, distribution and sale of the product, but allowing Sinclair to offer the product for sale in addition. This method provides the software house with an opportunity to increase its sales, as the product will be promoted in all Sinclair Mail Order literature. As orders are received, they will be passed to the software house, and Sinclair will require a percentage commission on orders generated in this way. Under this option, Sinclair packaging is not used for the product and so it remains very much the software house's 'own brand'.

Further details of the above options are given later on in this section, but first, the procedure for offering software to Sinclair is dealt with, together with Sinclair methods of review and appraisal.

19.1 How to offer a Product to Sinclair

When a software house offers a product to Sinclair for publication, two main areas have to be examined.

The first of these areas is the product concept. Under this heading, answers must be provided to such questions as:

What is the product?
What does it do?
For what type of market is it intended?
Does it exist?

If it exists:

How is it selling?
Methods of sale?
Volumes to date?
What machine does it run on?

If it does not exist:

What kind of sales are anticipated?
Based on what kind of information?
Are there any other products like it and if so which?

Obviously, some of the questions listed above assume that the product does not already exist for the QL or any other Sinclair computer. However, if it does run on some other computers, the second area to be examined would be concerned with how the product might be adapted to make use of the QL's features.

Specifically:

How would the product change?
What kind of pricing structure is envisaged?
What volume of sales are expected with respect to a low-cost computer such as the QL?
Would the target market change at all and if so, how?

Apart from considering the two areas described above, the product would need to be reviewed by Sinclair. For such a review to take place, the software house would need to send either:

1. The product itself, running on the QL, together with draft documentation. It need not be finished and completely bug free, so long as it is sufficiently complete to be able to be put out for review.

   or

2. The product running on another machine, preferably on Apple II, Macintosh or an IBM PC.

   or

3. A detailed product proposal on paper if the product exists only as a design. Such a proposal should cover at the very least the product concept and the proposals for the QL version of it.

19.2 Where Software Products should be sent for Review

1. Business software or proposals should be sent to the Business Software Editor, at the address given in the introduction to this manual.

2. Educational software or proposals should be sent to the Educational Software Editor, at the same address.

3. Anything that does not fall clearly into either of these two categories (e.g., games, compilers, utilities, expert systems etc.), should be sent to the Software Manager, at the same address.

19.3 How Products are reviewed and what Sinclair are looking for

1. Games and entertainment software

   Software of this type is generally reviewed by outside reviewers, often sixth formers. They are looking for originality, graphics, excitement, variety and pace. The software is judged under these five categories. The reviewers also compare the software to other similar products, and finally try to identify any bugs which may require fixing together with any improvements which may be made.

   As the computer games market is both extremely competitive and overcrowded, Sinclair can only consider absolutely top quality products for distribution. At the same time, the QL has expanded the range of possibilities in the context of entertainment software, thus any new ideas for using computers at home for entertainment and leisure activities would be reviewed with great interest.

2. Compilers and utilities

   Technical products of this kind will be reviewed internally in the first instance by Sinclair software engineers. They will judge the product for its completeness, the adequacy of its documentation, the speed at which it runs on the QL and its technical competence. In some cases where the product is of a very specialist nature it would be put out for review by an independent consultant.

3. Educational software

   Educational products, either for school, polytechnic, university or home use, will be reviewed internally at first, and possibly also by Sinclair educational consultants. The following categories are of particular interest to Sinclair:

   1. Software which caters for specific university and polytechnic markets.

2. Software which provides adult home education in fields previously uncatered for.

3. Software which actually teaches rather than tests foreign languages such as French, German or Spanish.

4. Software which teaches people how to expand their potential for different employment markets, for example, teaching touch typing, word processing, how to understand accounts, how to program etc.

5. Expert systems and authoring systems, especially if they have application software running under them which can also be sold.

4. Business software

   Business software will be reviewed internally unless it caters for a specific vertical market in which case Sinclair may seek permission to have the product reviewed in detail by an independent consultant. When possible business packages are being considered, both the company and the product will be examined very carefully. Thus the following are particularly sought after:

   1. Established suppliers of business products with a respected name in the business market.

   2. Products which would benefit from distribution in wider markets and at a lower price than at present.

   3. Suppliers who can, if necessary, provide any direct support needed by their product to Sinclair customers, possibly at additional cost.

   4. A secure financial backing which will ensure that the company will not disappear after Sinclair have launched the product, leaving no support for it.

19.4 Contractual Options in dealing with Sinclair Research

In the introduction to this section several possible contractual options were described, which will now be explored in more detail.

1. Distribution in Sinclair packaging

   Royalty contract – every software house which offers a product to Sinclair Research for distribution in Sinclair packaging under the Sinclair name, will be asked to sign a Licence Agreement of the type shown in Appendix A. This agreement allows for the grant of an exclusive licence to Sinclair Research for the distribution and sale of the specified software products, in return for a royalty which is normally 20% of the selling price.

2. Distribution of finished goods

   Those software houses from whom Sinclair agree to buy a complete finished product packaged to Sinclair specifications, will be asked to sign a second contract in addition to the software licence contract described above. This second contract would provide for the supply and purchase of manufactured goods on an 'at cost plus' basis. In this way, a packaging specification would be agreed upon for the product, and Sinclair would nominate approved suppliers of each component of that packaging. The software house would then purchase these components from the nominated supplier at a price previously negotiated between Sinclair and the supplier. The cost of the product would then be passed on to Sinclair, the software house having added a fixed margin as their handling fee for controlling production.

3. Sinclair approved products

   Under this option, the product would neither be sold in Sinclair packaging nor would it carry the Sinclair name. It would instead be packaged in the software house's own packaging under its own name. It would, however, be promoted as a Sinclair endorsed product in the Sinclair catalogue. Orders would be sent to a special PO box at Sinclair's despatching warehouse and would then be forwarded direct to the software house for fulfilment. Sinclair would, of course, expect to be paid a percentage commission on orders generated in this way, which would normally be equivalent to 15% of the retail selling price.

19.5 Promotion and Distribution

1. Sinclair packaged software

   As might be expected, software carrying the Sinclair logo attracts the bulk of Sinclair promotional activities. In particular, all software carrying the Sinclair logo and name will:

   1) be offered initially to all Qlub members directly, possibly at a small discount as an introductory offer;

   2) be carried in a catalogue which will be included with every QL shipped;

   3) be launched to the trade and specialist press, and included in advertising campaigns from time to time;

   4) be the subject of special promotions which will be considered for vertical market software;

   5) be offered the possibility of consideration for bundling contracts from time to time. This can be a very lucrative way of ensuring that the software reaches the widest possible market;

   6) be offered to our local area offices and distributors all over the world, for translation into foreign languages;

   7) be similarly offered to our Boston Office for publication and distribution in the United States.

2. Sinclair endorsed products

   Where a product is not distributed in the Sinclair packaging, but is being promoted and offered for sale through Sinclair, then it is likely to be promoted using methods 1.) and 2.) only, though from time to time, where appropriate, other methods of promotion and marketing will be considered. To attract the full range of Sinclair's marketing activities a product needs to be offered for distribution in both the Sinclair packaging and brand name.

19.6 Summary

Many software houses writing software for personal computers today are concerned about the possible dilution of effort that is entailed when a product has to be packaged, promoted, marketed and sold as well as developed. Sinclair Research are known for their ability to obtain media coverage and for their marketing and distribution capabilities.

In the case of the QL, Sinclair believe that software houses can be offered distribution opportunities without equal. The Qlub will enable direct contact to be retained with customers on a far larger scale than previously possible with other Sinclair computers. It is proposed to use the Qlub Newsletters as a method of informing customers of every new product launched in advance of the general public. Small discounts will be offered which will make the product attractive to the customer, but will not begin to approach the kind of discounts Sinclair would need to give should the product be offered through a distributor or a retailer.

It is hoped that software houses will feel that to offer software to Sinclair in one of the ways described above will prevent many of the problems previously associated with bringing their products onto the market place.

20.0 Bibliography

1. MC68000 16/32-bit microprocessor programmer's reference manual.

   Published by Prentice–Hall for Motorola. ISBN 0-13-566795-X.

   Contains instruction set details for the MC68000 and MC68008, including permissible addressing modes and bus cycle diagrams. Some hardware detail is included, but no timing diagrams.

2. Motorola Semiconductors 16-bit microprocessors data manual – 1983.

   Published by Motorola Ltd., York House, Empire Way, Wembley, Middlesex.

   Contains the hardware reference for the MC68008.

21.0 Index