PART 2 QL System Procedures

3 THE QDOS PACKAGE

'The whole world is in
a state of chassis.'

Sean O'Casey

Conceptually, QDOS is the chassis, and you are the coach-builder. It is important to view the QDOS system in this way. There are perhaps many programmers who are accustomed to thinking of operating systems as resource allocation programs, under which their own application programs run. Strictly speaking this is not true of QDOS.

QDOS is a chassis of procedures. Application programs which you may write are at liberty to use any of these procedures. Furthermore QDOS has the appropriate hooks to enable its collection of procedures to be expanded or modified as required. It is this structure which provides the assembly language programmer with so much flexibility.

3.1 System memory map

The physical memory map of the Sinclair QL, and the RAM map imposed by QDOS, are so important to the assembly language programmer that we will look at these first. Figure 3.1 shows the layout of the two maps. Map 1 is the physical memory map of the microcomputer, and map 2 is the map of the RAM.

PHYSICAL MEMORY MAP

The total amount of memory that can be accessed is 1 Megabyte. The system ROM, together with the address space for the plug-in ROM, occupy the bottom 64 Kilobytes ($00000 to $0FFFF). The next 64 Kilobytes ($10000 to $1FFFF) are dedicated to I/O devices. Only 16 Kilobytes of this area are currently allocated. Above the first I/O block lies the RAM. The RAM always has a base address of $20000. The top of the RAM area will, on a standard 128K machine, be $3FFFF. With the 0.5 Megabyte expansion RAM module plugged in, the top of RAM becomes $BFFFF. The final 256 Kilobytes of memory address space is reserved for additional I/O. This final area, together with the I/O area that exists further down the map, supplies the user with a total of 304 Kilobytes of expansion I/O. This may seem rather large but it serves to act not simply as device address space, but also as device driver program space. An advantage is clearly evident here because it means that no RAM space needs to be taken up in the process of adding additional I/O facilities.

QDOS RAM MEMORY MAP

Now that we know how the memory space is divided up physically, let us look at how the RAM space is allocated. The bottom 32 Kilobytes ($20000 to $27FFF) are dedicated to the screen display. The remaining 96 Kilobytes ($28000 to $3FFFF), or 608 Kilobytes ($28000 to $BFFFF), are managed by QDOS in the form of five major areas. There are a number of system variables that are used to determine, at any one point in time, the sizes and free RAM pointer values of these QDOS areas. The variables, their use, and their absolute position in the memory map are shown in Fig.3.2. Each variable is stored as a long-word, i.e., is 32 bits in length. The mnemonics given to the variables (e.g., SV_BASIC) are for reference only - they will not be recognized either by SuperBASIC or QDOS.

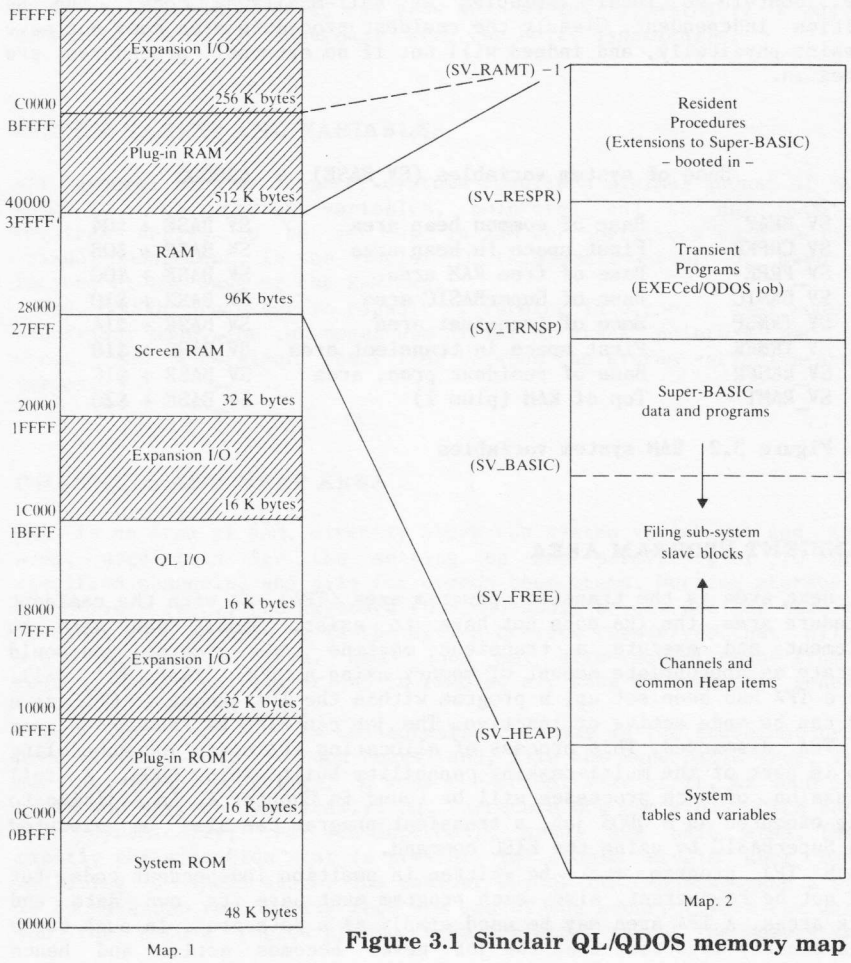

Figure 3.1 Sinclair QL/QDOS memory map

RESIDENT PROCEDURE AREA

At the very top of the available RAM are the Resident Procedures. These procedures, and any tables that may accompany them, are loaded into memory when the system is booted (i.e., reset). The only way to change the area allocation after it has been set up is to re-boot the system. More will be said of this in Chapter 8. The term resident procedure area is used because this is an area that, once booted in, will be permanently resident. Also the most common use of this area is the storage of user-defined SuperBASIC procedures. By adding the entry point names of the procedures into the procedure name list of the SuperBASIC interpreter utility, the procedures themselves will become extensions to the SuperBASIC language.

Any procedure added to the system in this way must be re-entrant (i.e., contain no local variables or self-modifying code), and be position independent. Clearly the resident procedure area does not have to exist physically, and indeed will not if no external procedures are booted in.

Base of system variables (SV_BASE) = $28000

SV_HEAP	**Base of common heap area**	**SV_BASE + $04**
SV_CHPFR	**First space in heap area**	**SV_BASE + $08**
SV_FREE	**Base of free RAM area**	**SV_BASE + $0C**
SV_BASIC	**Base of SuperBASIC area**	**SV_BASE + $10**
SV_TRNSP	**Base of transient area**	**SV_BASE + $14**
SV_TRNFR	**First space in transient area**	**SV_BASE + $18**
SV_RESPR	**Base of resident proc. area**	**SV_BASE + $1C**
SV_RAMT	**Top of RAM (plus 1)**	**SV_BASE + $20**

Figure 3.2 RAM system variables

TRANSIENT PROGRAM AREA

The next area is the transient program area (TPA). As with the resident procedure area, the TPA does not have to exist. Should you wish to implement and execute a transient machine code program, you would allocate an appropriate amount of memory using a QDOS 'TRAP #1' call. Once a TPA had been set up, a program within the TPA (known to QDOS as a job) can be made active or inactive. The job can also be suspended, or the TPA discarded. This process of allocating TPA space and scheduling jobs is part of the multi-tasking capability built into QDOS. A full discussion of such processes will be found in Chapter 4. In addition to being executed as a QDOS job, a transient program can also be executed from SuperBASIC by using the EXEC command.

Each TPA program must be written in position independent code, but need not be re-entrant. Also, each program must have its own data and stack areas. A TPA area may be used simply as a data area. In such cases it is clearly important that the job never becomes active and hence

executed! Owing to the fact that jobs can be created and discarded almost at will, the total TPA area will grow and shrink dynamically. Note, however, that any one TPA program will always take up a pre-declared amount of space.

SUPERBASIC AREA

This area contains all currently loaded SuperBASIC programs and all related data (i.e., both program data and SuperBASIC interpreter utility data). Clearly there is no way of telling, a priori, how much space is going to be required by a SuperBASIC program. In view of this, QDOS makes a special allowance for the SuperBASIC area and permits it to grow and shrink dynamically. The total transient program area, immediately above the SuperBASIC area, can grow and shrink dynamically also, and therefore the entire SuperBASIC area can shift dynamically.

SYSTEM TABLES AND VARIABLES

All general purpose computer systems require a minimal amount of RAM in which to store important variables, pointers, and so on. QDOS also requires tables to be set up in RAM for operations such as job and channel management. In the Sinclair QL this small amount of memory is located at the base of the RAM map.

The 68000 processor is capable of running in either a user mode or a supervisor mode. The two modes use different stack pointers and stack areas. The supervisor stack lies between the system variables and the tables.

CHANNELS AND HEAP AREA

This is an area of RAM, directly above the system variables and tables area, used both for the setting up and permitting of I/O through specified channels, and also for common heap items. Working storage for I/O drivers (e.g., the keyboard routine) would be one use for this area. In cases such as this it is the device drivers themselves that allocate space within the area. QDOS jobs may also request space from this region. When a particular job is removed any heap allocations owned by it will be removed also.

This area of RAM is conceptually the same as the transient program area (in that they are both heaps) and, like the SuperBASIC area, it is not possible, a priori, to know the actual size of it. As such it too grows and shrinks dynamically. So, there are now two areas which vary in size dynamically; the TPA+SuperBASIC area, and this area. This is exactly the situation that is present in simple single user systems utilizing a single stack. The easiest way of implementing such a system is to have one region grow from one end of the memory toward the middle, and the second region growing from the other end of the memory toward

the middle. When the two regions meet you have run out of memory! Figure 3.1 shows that the same form of implementation is used in QDOS. The TPA+SuperBASIC area grows downwards and the channel and heap area grows upwards. Any memory left in the middle of these two areas is given over to filing sub-system slave blocks.

FILING SUB-SYSTEM SLAVE BLOCK AREA

This area exists between the dynamically variable SuperBASIC and channel and heap areas. All the remaining RAM, at any one point in time, is given over to filing sub-system slave blocks. These slave blocks are invisible to the user and merely duplicate data held on the Microdrives. Their use enables QDOS I/O to make Microdrive accesses as efficient as possible. The bigger the amount of free RAM, the greater will be the efficiency of Microdrive accesses. This mechanism means that QDOS is constantly using all available memory to its greatest advantage.

3.2 Bootstrapping

When the system is first turned on, or a reset is performed, execution will start at the base of the system ROM. Once the system variables have been determined, and a RAM test carried out, a system scan will be performed to find out the true configuration of the machine.

First the plug-in ROM address map will be checked (at $0C000) for the characteristic long-word '$4AFB0001'. If this word is found it is assumed that a ROM exists and that it contains appropriate code. Next the expansion areas are checked for device drivers. Assuming control is returned to the bootstrap routine, an attempt will finally be made to open either a device called 'BOOT', or the file 'MDV1_BOOT'. If this attempt is successful then the respective file will be loaded into memory (as a SuperBASIC program) and executed.

3.3 System calls and utilities

There are two major types of routine that assembly language programmers can access from within their own application programs and subroutines. The first type is that of 'TRAP #n' calls made to QDOS procedures; the second are those general utilities that are accessed through vectors.

QDOS ROUTINES

System calls to QDOS may either be treated as atomic or partially atomic. Most QDOS routines are atomic in nature. Atomic routines are executed with the 68000 processor in supervisor mode. In this mode no other job can take priority over use of the processor and, therefore,

the routine will be executed from start to finish before being 'swapped-out'. Note that this is the general case only; the routine could be interrupted by an interrupt service procedure. Routines which are partially atomic will complete some sort of primary operation, but will then allow another job to swap-out the original calling process until a later moment in time. All the I/O calls are partially atomic unless specifically accessed as being fully atomic. Scheduler calls are partially atomic.

Note that executing a TRAP #0 instruction, on the QL, will force a switch to supervisor mode. No registers will be altered (except, of course, the stack pointer, which will become the SSP). User mode can be re-entered simply by altering the status register.

QDOS procedures are accessed via 'TRAP #n' calls with register D0 indicating which particular call is required. Chapters 4 to 6 describe these trap calls in detail, but some generalities are worth mentioning at this point. Register D0, as well as containing the procedure index on entry (as a byte), is used also to return an error status (as a long-word) to the calling process. If the error code returned is not zero then an error has occurred. Small negative error codes are used to indicate standard errors. These error codes are listed in Appendix C. If the trap call invoked some form of additional device driver, the error code returned can be a pointer to a specific error message. In order that the two types of error return code might never be confused, the pointer type error code is in fact a pointer to an address $8000 below that of the true error message. Potentially, all QDOS routines can return the error 'ERR.BP' (-15), signifying 'bad parameter'.

In addition to the use of register D0, data registers D1 to D3 and address registers A0 to A3 are variably used to pass values to and from the QDOS procedures. When the appropriate registers have been set for any one call the appropriate routine is accessed by simply executing the appropriate 68000 trap instruction. For example, to suppress the cursor in the window belonging to channel ID $10001, the following may be used:

```
:
:
move.b      #15,d0          ;Suppress cursor routine
move.w      #0,d3           ;Return immediately
move.l      #$10001,a0      ;Channel ID
trap        #3
:
:
```

The full description of this QDOS routine, given in Chapter 6, shows that it is capable of returning three errors (the two listed and the more general 'bad parameter' error). It would of course be wise to check for these errors after the trap call has been made.

UTILITY ROUTINES

These routines are, as far as this text is concerned, a mixture of simplified trap routines and SuperBASIC utility routines. Each routine

is discussed in detail in Chapter 7. By using these routines the assembly language programmer can greatly simplify basic I/O code and can even incorporate floating point calculations into application programs!

The method of accessing either type of utility is the same, and simply involves setting up the appropriate call parameters and then performing a subroutine call to the required vector. For example, to send out the ASCII representation of an integer word in the memory location labelled 'RESULT' to the command channel (#0), the following could be used:

```
        :
        :
        move.w      result(PC),d1     ;Get result
        sub.l       a0,a0             ;Select channel zero
        move.w      #$CE,a4           ;Convert and print routine
        jsr         (a4)
        :
        :
result: defs 2                        ;Integer result register
        :
        :
```

Once again it would be normal to check for all possible error return conditions.

There are four Microdrive support routines available that have to be handled in a slightly different way. First, their access vectors actually point to an address $4000 bytes before the true entry points. It is important therefore to add on this offset when making the vectored calls:

```
e.g.    move.w    md.verin,a4
        jsr       $4000(a4)
```

Second, they do not return an error code. Instead, they have multiple return points.