3 QDOS – AN OVERVIEW

3.1 Introduction

The aim of this chapter is to introduce the fundamental concepts behind QDOS. Detailed explanations of particular aspects are left until the reference chapters in the latter half of the book.

The QL is distinguished from other computers by the fact that it supports *multitasking* of Jobs. In plain English, this means that several completely independent programs can all be running together.

A little thought will reveal that this method of operation is not as complex as it might at first seem. For example, you may wish to play against the QL at chess. Moves could be made on a one per day basis. This would allow the QL plenty of time to run through very complex move analysis, but would not allow the computer to be used for anything else all day. Now the real use of *multitasking* Jobs can be appreciated. If the chess playing program is set up as a Job with a low priority, it can operate in the *background* all day. Other programs such as business applications, games or other general tasks can be run at a higher priority. The QL then uses the processor for playing chess only when it is not required for the foreground task (such as when the foreground task is waiting for a key to be pressed).

Another more trivial example of multitasking is that of a clock. The time and date can be set when the QL is first switched on. A Job for producing a clock display on the screen can then be loaded in. This Job can be run at a very low priority, so the clock's time will be updated on the screen at fairly regular intervals. The nice feature of running a clock in this way is that it just looks after itself. Basic programs can be run quite happily without ever knowing that the clock is there.

To summarise, a Job is a program unit which can operate in isolation from other program units in the QL. The Job spawning program in section 4.4.2 illustrates how a large number of Jobs can usefully coexist.

You may now be thinking how terribly complex it must be to get all of these Jobs running happily together. In other less powerful systems, this would indeed be a problem, and would involve messing about with intercepting interrupts. However, on the QL it is not too complex at all – QDOS takes care of all the difficult bits. To ensure that all Jobs coexist peacefully, QDOS has to ensure that they don't use the same areas of memory for different purposes, and that the allocation of machine resources is carefully controlled. For example, if two Jobs wanted to output data to a printer at the same time, garbage would be produced if output from one Job was interleaved with output from another Job. To remedy this the last Job to request the printer must be forced to give way to the one which is already using the printer. QDOS hides all of these problems from the programmer, and ensures that the second Job is queued for output and then left alone until the first Job has finished.

Rule number one can now be understood. That is **USE QDOS AT ALL TIMES AND DON'T WRITE PROGRAMS WHICH ARE NOT PROPERLY CONNECTED INTO THE SYSTEM**. What this means is that QDOS should always be allowed to allocate memory, printer channels etc., so that it knows *everything* which is going on. If the user were, for example, to POKE a 68008 program directly into some empty part of memory, this could be disastrous if QDOS doesn't know about it. Why? Because the memory map of the QL is continuously changing, so if QDOS doesn't know what's in a particular area, a POKEd in program may be allocated as if it were free memory! All of the available free memory is liable to be used as a large file buffer when writing to or reading from microdrives. QDOS must therefore be told whenever a Job is set up, or when memory is used directly.

Provided that QDOS is in full control of the memory and other machine resources, Jobs will operate in the multitasking environment. The three sections of QDOS involved with the system control are memory management (making sure that Jobs don't overwrite each other's memory), resource allocation (making sure that resources like the printer are only used by one Job at a time) and scheduling (deciding which Job should be allowed to use the 68008 central processing unit at any one time).

3.2 QDOS and System control

3.2.1 Memory management

The introduction section pointed out that some form of memory control is required. Without good control, there will be many Jobs all running at the same time, writing over each other's memory. One of the major requirements of QDOS is that it should look after the QL's memory and ensure that no two Jobs ever use the same memory by accident. Of course, this doesn't mean that you can just produce a *buggy* program and expect QDOS to protect the rest of the machine! POKEing at random into the QL's memory will lead to an unavoidable crash. QDOS is only able to prevent operational programs which adhere to the rules from clashing. With this aim in mind, the available memory is split up into several distinct sections (refer to figure 3.1).

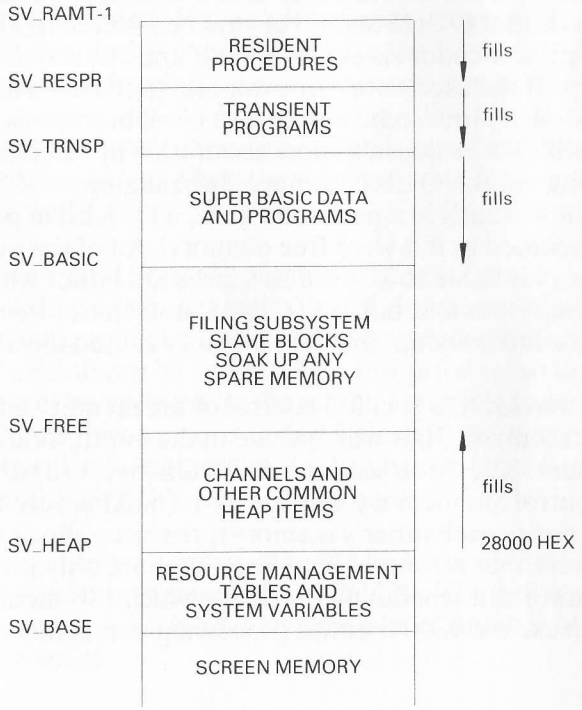

Figure 3.1 – Memory map as seen by QDOS

Uses of memory – a top down approach

The Resident Procedure area

This block of memory is allocated when the machine is powered up, and once allocated remains fixed. All memory between the top of available RAM (SV.RAMT) and the bottom of the area (SV.RESPR) is devoted to resident procedures. The type of programs which would normally be put in this part of memory are those which will be required throughout the time that the machine is being used. Note that SV.RAMT will vary from machine to machine depending upon the amount of extension memory which is plugged in.

Resident procedures which add into SuperBASIC should be placed in this area. The procedure (or function) names can then be linked into the BASIC name list, so they will always be accessible from BASIC. A couple of examples covering procedures and functions which link in can be found in section 10.8. It is worth noting that all procedures which are defined in this way MUST be *re-entrant, position independent* and MUST NOT contain any *self modifying code* or *local variables* (the glossary explains what all of these terms mean).

The Transient Program area

As the name implies, this area contains programs which are not resident, and which might be removed at some stage. Most Jobs are put into this area of memory. The transient programs are loaded immediately below the resident procedures. Each transient program must be self contained (code, data area and stack). Any programs which are not relocatable (ie. which must be loaded at some specific address) have to be loaded by a special *linking loader*. This is because transient programs are normally loaded using the EXEC command and then activated as Jobs. Loading by this method means that a particular Job cannot be guaranteed a particular position in memory.

If the transient program area is used for other purposes than holding Jobs, such as storing data, care must be taken that the *data* is not activated. Such activation would lead to the machine crashing!

The SuperBASIC area

All data related to the correct operation of SuperBASIC is stored in this area. Included is the typed in BASIC program and all of the BASIC system variables.

It is important to relise that the whole of this area of memory is continually moving about. It moves down in memory whenever more transient program area is allocated. It moves up if transient program area is released. Part of the area moves when new lines are typed in on the keyboard, and the size changes as new data is created during the running of a BASIC program.

The whole of the QL's memory map is in a constant state of change. This is quite different from most other small computers, where items like BASIC variables are stored in fixed locations. Whilst this may at first sight seem like a very odd way of doing things, in practice it is very neat. The 68008 can simply use one of it's eight address registers (A6 for BASIC) to point to the bottom of any area. All operations within that area are then carried out *relative* to this address register, creating the illusion of all data being fixed in position.

Free space (used as filing subsystem slave block)

This area stretches from below BASIC down to the top of the channel and heap area. Normally, when files are not being accessed, this area is empty. However, when QDOS is instructed to read from a file or write to a file, it is brought into use.

Microdrive accesses are inherently rather slow. Files are normally quite long, and are held in more than one sector on the cartridge. If all of the required data could be loaded into memory at once, rather than doing lots of slower accesses of individual sectors, this would clearly be advantageous. Although the user doesn't normally see this background operation, this is in fact what QDOS does. Maximum use is made of the available memory, so copying files on an empty machine will therefore be very efficient. Once the machine gets full of other programs (like a large BASIC program), the available free space in this area becomes much smaller. Only a little data can then be buffered in memory, and microdrive copying would consist of transferring lots of small packets of data from one drive to the other – very inefficient.

The system heap

This area of memory is used to store channel definitions, data, working storage for the I/O subsystem etc. The area can be expanded or reduced in size via the relevant manager traps. Jobs can allocate memory in this area for their own use. The system is designed so that all of the memory used by a Job will be released for use by others if the Job is deleted. The system heap is covered in much greater detail in section 5.2.4.

System variables, system tables and the Supervisor stack

This fixed area of memory resides just above the screen memory. The Supervisor stack is used by routines which are running in the Supervisor mode on the 68008. The system variables are listed in Appendix M.

3.2.2 Resource allocation

The process of resource allocation is concerned with all of the devices which can be addressed by Jobs. The screen, microdrives, keyboard, sound generator and serial printer are all resources. Clearly, if more than one Job wants to use the same device at the same time, such as the microdrive for saving a file, only one Job is going to be lucky! Ensuring that the first Job gets priority, and that the second Job is put into a waiting mode until the first has finished is all a result of correct resource allocation.

The process of allocating resources is controlled by manager traps (chapter 5) and I/O allocation traps (chapter 6).

3.2.3 Task scheduling

There are several aspects of task scheduling in a multitasking machine. The 68008 CPU can only cope with one Job at a time. If any Job were allowed to run without the scheduler, there would be no way of stopping it from using all of the CPU's time itself. This is where the scheduler comes in.

Whenever a Job is created, it is given a *priority*. This priority determines what share of the CPU time is allocated to that particular Job. A Job with a low priority of 1 will only get the use of the CPU for a tenth of the time given to a Job with a priority of 10. Job priority can be anything from 0 (in which case the Job never gets any CPU time) up to 127. The BASIC command interpreter is run as Job 0. This normally has a priority of 32.

Jobs can exist in one of three states:

Active Jobs have a priority between 1 and 127. They obtain a share of the CPU resources determined by the priority as explained above.

Suspended Jobs are ones which are capable of running (ie. priority is between 1 and 127), but which are not running because they are waiting for another Job or I/O. The scheduler will automatically reschedule a suspended Job as soon as the Job which it was waiting for has finished, or as soon as I/O becomes available. Jobs can be suspended indefinitely, or for a specific period of time (see section 5.4.2). This is a very useful feature of QDOS. In older systems, a program waiting for I/O would waste all of the CPU's time just trying to transfer a small amount of data.

Inactive Jobs are ones whose priorities have been set to zero. An inactive Job does not get a share of the CPU resources.

The scheduler is usually invoked by a hardware interrupt some 50 times every second. It is also invoked whenever Jobs are activated (see MT.ACTIV). Each Job is allocated a *temporary priority*. This priority is incremented by the assigned priority of the Job on every scheduler call. A check is then made of the temporary priority of each and every Job. The one with the highest temporary priority is allowed to have control of the CPU until the next scheduler call. The temporary priority of the Job which was running is then reset to zero before incrementing all the temporary priorities again.

Some programs with parts which are time critical (such as reading in a sector from a microdrive) must **not** be interrupted by the scheduler. A routine which must operate to completion in one time slice is called *atomic*. Most of the system calls are atomic. It

is possible to ensure that user routines are *atomic* by running them in *supervisor* mode on the 68008 (but take care not to call a 'partially atomic' system routine from within an atomic user routine – the scheduler may be turned on again by partially atomic routines).

3.3 Input and output to Jobs

QDOS contains extensive facilities for communication between Jobs and devices, whether these are standard or expansion. Provided that all communication is carried out through QDOS, no clashes between devices will occur. Section 3.2.1 explains how Jobs are designed to interact correctly by allocating resources properly.

Input and output are covered by the I/O utilisation traps (chapter 7) and some of the special utilities (chapter 8). The facilities provided are sufficient for most applications. The screen for example is covered by routines to set up windows, define output channels into these windows, control the colours, size of text, type of cursor, graphics plotting and filling, border producing etc. In the unlikely event of these routines being unsuitable for a particular program, it is even possible to incorporate special user defined routines which will work in conjunction with QDOS. The main criterion behind all of these is that QDOS must know what is being used at any time, even if it isn't actually a resident QDOS routine which is doing it.

To this end, channels can be set up in a standard format which QDOS understands. If a channel is OPENed or CLOSEd using such a standard format (called a channel definition block), QDOS will be able to keep track of exactly what's going on.

3.4 Support for user programs

It should now be appreciated that QDOS provides a very powerful kernel which looks after the general running and operation of the QL. This is the primary task of QDOS.

The ROMs inside the QL are however not soley concerned with system operation. There are a considerable number of useful utility routines which can be accessed from user code. All of these are covered in the utilities chapter 8.

Perhaps the best understood of these utilities is the BASIC command line interpreter. This is a Job in its own right (a rather special one), but is not an essential part of the QL. Other Jobs can run quite happily in the absence of BASIC. Many of the routines which are used by BASIC are equally applicable to user code. The arithmetic package is a very good example of this. The example in section 8.5.5 illustrates the point very nicely by producing an extra BASIC function to evaluate SINH of a variable.

Other simple routines for serial input and output, number conversion and microdrive handling are just a few out of the wide selection which are documented in the latter parts of this book.

4 Experimenting with QDOS

4.1 Introduction

Most of the other chapters in this book are devoted to describing how QDOS operates. This chapter is different because it is devoted to experimentation both from BASIC and Assembler. If you haven't read chapters 2 and 3, now would be a good time to do so, since this chapter assumes a certain basic understanding of the system and concepts like 'traps'.

Both the QL experimentor program and the examples in this section have been produced to give users an *intuitive feel* for the QL system. The QL Experimentor allows many of the features of QDOS to be accessed. Initially, the Experimentor program will allow traps to be generated from BASIC. It is essential to grasp the concepts behind the use of traps on the QL if other exciting features of QDOS are to be exploited. As we delve deeper into the depths of the computer, it will become apparent that the more advanced features can only sensibly be used from Assembler.

Practical programs written in Assembler are introduced at this stage to illustrate the more important aspects of QDOS, as seen from a machine code program. As a simple introduction, a Job is set up to display the time on the screen. Such a simple program is very useful to analyse because it involves the initialising of a Job, setting up a window, and converting the time and displaying it on the screen. The *self-cloning* Job example then illustrates many of the more complicated aspects of Jobs, including memory allocation and Job control.

There are more specialised examples scattered throughout the book. Chapter 10 contains complete examples for producing a BASIC FUNCTION and a BASIC PROCEDURE in 68000 Assembler.