5 The Manager Traps

5.1 Introduction to manager traps

This chapter deals with the allocation of available resources on the QL. If the introduction chapter on QDOS (chapter 3) hasn't already been read and understood, then now is the time to do so.

Available resources on the QL fall into two areas. The first is the memory, and the second consists of all the other pieces of hardware which are available for use on the QL, including the 68008 CPU itself. All types of program which can be used on the QL need memory in which they can be stored and memory in which they can store data associated with the task in hand. A selection of routines are therefore available to allocate memory to programs, both for use as program and/or data storage. These routines are dealt with under section 5.2. Another selection of routines are available for management of hardware devices other than the memory. These devices are either standard built in components such as the real time clock, IPC (intelligent peripheral controller), or external devices such as hard discs or parallel printer ports. The internal hardware is dealt with by a series of specialised *device drivers* which are built into QDOS. External hardware can easily be linked into the system by a number of well defined routes. This aspect of system management is explained further in section 5.3.

Generally, most of the system resource management is carried out by the manager traps, which are all accessed via trap #1. A certain amount of resource management can also be effected by some of the vectored utility routines as described in chapter 8. These can be used by device drivers called from Trap #2 or Trap #3, but should NOT be used directly from user code.

5.2 Memory management on the QL

Memory management often concerns both the protection and allocation of memory. In the QL, only allocation is catered for. In practice, this means that the entire system can be crashed simply

by POKEing garbage into any area of memory used as QDOS workspace (on a more sophisticated hardware system it would not be possible to do this). Since important blocks of program and data can exist anywhere within the QL's memory, it is *essential* that memory is only used in the manner required by QDOS. A number of very powerful traps are provided to deal with the proper allocation of memory. The particular type of allocation depends upon the exact demands which are placed on the program and/or data to be stored in the memory. The following sections give an introduction to the different types of memory allocation which can be used.

5.2.1 Resident procedures and their memory

Resident procedures were introduced in chapter 3 as a very powerful form of program. They sit at the top of the memory map (see figure 3.1). The reason why they are such a powerful a form of program is that the memory in which they sit can normally only be allocated when the QL is booted (powered up). Once the resident procedure area has been allocated, it cannot usually be either deleted or extended. The reason for this is that the transient program area is immediately below the resident procedure area. The transient program area cannot be relocated, and cannot therefore be moved, as would be required by a change in size of the resident procedure area. Only under the excetional circumstance of there being no transient procedures in the QL is it possible to change the resident procedures area.

5.2.2 Jobs as transient programs

As explained in chapter 3, the transient program area of memory is dynamically alterable. New programs can be added (in which case the area will expand) or old programs can be removed (in which case the area will contract). The programs which can be introduced into the transient program area are usually referred to as *Jobs*.

A considerable number of events can befall a Job during its life in the QL. Some of these are essential and inherent in the fact that the Job exists (ie. Job creation), others are optional (such as changing a Job's priority or deleting it).

The area of memory allocated to a Job can normally be considered in two parts. The code for the Job occupies the first part of memory allocated to it. The data used by the Job occupies the rest of the allocated area from the top of the code to the top of the area. These areas are specified when the Job is initially created. At creation the Job can be defined as an independent unit in its own right, or as a sub-unit of another Job. Jobs which are *owned* by other Jobs will be removed if the owner Job is removed.

To enable the user to keep track of all these Jobs, each one is given it's own unique marker called the *Job ID*. The first Job to be created is Job 0, and is always the BASIC command line interpreter. Subsequent Jobs are allocated their own Job number. The numbering of Jobs is entirely automatic, and the number of any particular Job will always depend upon the number of other Jobs in the system when it was created. The bottom word of the Job ID contains the Job number and is used as an index into the Job table. The top word of the Job ID is a *tag*. Each new Job is labelled with a Job tag which is one greater than the last tag allocated. Hence, tag 1 is followed by tag 2, 3, 4 etc.

Once a Job has been created, there are three well defined states into which it may fall. It can either be *active*, *suspended* or *inactive*.

An *active* Job shares CPU resources with other Jobs. In order to determine how much of the CPU time should be given to any particular Job, each one is assigned a priority of between 1 and 127. The highest priority is 127 and the lowest (for an active Job) is 1. The scheduler decides which of the currently active Jobs should have the CPU at any particular time.

A *suspended* Job is one which is still active. However, it is suspended to wait for input, output or another Job which isn't ready yet. The scheduler therefore doesn't assign any CPU time to a suspended Job until the process which it is waiting for is complete. The Job is then re-activated.

An *inactive* Job is one whose priority has been set to 0. No CPU time is allocated to such a Job. An inactive Job is still present in the transient program area, and could be re-activated at any future time by another Job.

Finally, a Job can be removed from the transient program area. There are two basic types of removal. The normal type will only allow a Job to be removed if it is inactive. Alternatively, a Job can be forcefully removed from the area (even if it is still active). On removal of a Job, any other Jobs that it owned are also removed.

5.2.3 Memory allocation for Basic programs

The Basic command line interpreter is set up as Job 0. As with any other Job, memory is required to store the program (in this case not 68000 code but a tokenised Basic program) and data. Unlike the other Jobs, which are given a fixed amount of memory, Job 0 is allowed to allocate more space for itself. NO other Jobs can expand themselves dynamically like this! Two manager traps (D0 = 16 and 17) are provided to expand or contract the memory available for Basic.

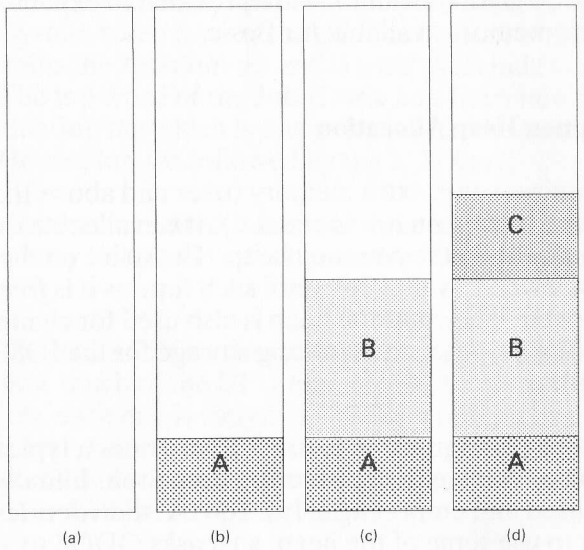
5.2.4 Common Heap Allocation

If a Job requires some extra memory (over and above that which was allocated to it when it was created), it can allocate the required amount on the common heap. The space on the heap will then be owned by that Job until such time as it is *freed*, or the Job is removed. The common heap is also used for channel definition blocks, data, and working storage for the IOSS (I/O sub-system).

The sequence of diagrams in figure 5.1 illustrates a typical progression of storage in the common heap area. Initially, the heap is unused and empty (fig 5.1a). Job 'A' then decides that it would like to use some of the heap, so it asks QDOS to allocate some using MT.ALCHP (trap #1 with D0=$18). The allocated area is shown shaded in fig 5.1b. Now Job 'B' comes along and asks for some heap. Again, QDOS allocates some memory to this Job just above the area owned by Job 'A' (see fig 5.1c). Another Job (Job 'C') now requests some memory as well. It is allocated some just above Job 'B' (see fig 5.1d).

At some later time, Job 'B' is removed from the system. QDOS has kept a record of who owns the various bits of memory in the heap, so it realises that the area of heap allocated to Job 'B' is not

needed any more. The area is therefore cleared so that it can be used by other Jobs (see fig 5.1e). Another Job (Job 'D') now comes along and requests some memory in the heap. QDOS finds that part of the area which used to belong to 'B' is suitable, so this is allocated to 'D' (see fig. 5.1f). Job 'E' has a larger requirement for heap area when it puts in a request. QDOS finds that the only space which is large enough lies above Job 'C'. This area is allocated to 'E'. Job 'A' has now reached a stage at which it needs some more memory. Again, it puts in a request using MT.ALCHP. QDOS searches through the heap and finds that 'A' can be fitted in between the top of 'D' and the base of 'C'. Consequently the new space is allocated there.



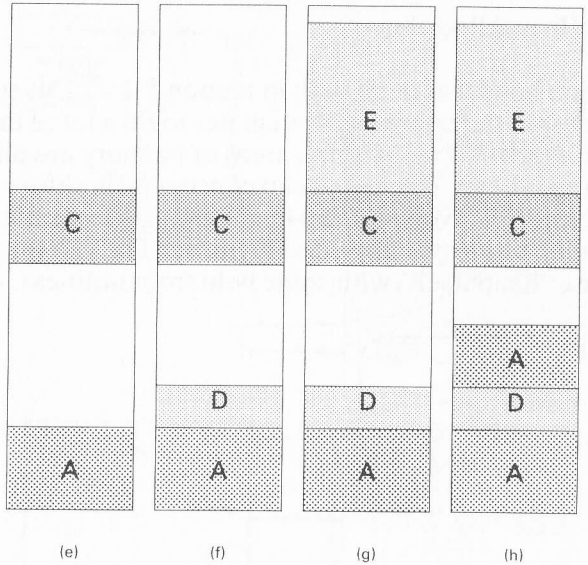


Figure 5.1 – Common heap usage

This process continues throughout the period while the machine is powered up. It will be appreciated that the whole of the heap will become terribly fragmented if Jobs are continually allocating, releasing and re-allocating space. There is a considerable overhead for each allocated space because the operating system has to keep 16 bytes of information on each section (so that it can get rid of allocated heap if a Job is removed). A much more efficient way of allocating heap storage is for each Job to take the maximum amount of memory which it will ever require. The Job can then set up its own *user heap* within this allocated area without bothering the operating system.

A number of useful traps and utilities are provided which will help user programs to manage their own heaps.

5.2.5 User Heap Allocation

The common heap was dealt with in section 5.2.4. This section pointed out that the operating system has to do a lot of extra *housekeeping work* if many small areas of memory are allocated in the common heap. The user heap concept is therefore very useful. It basically involves a Job in allocating a large area in the common heap or its own allocated memory. The Job then looks after the user heap itself (with some help from utilities).

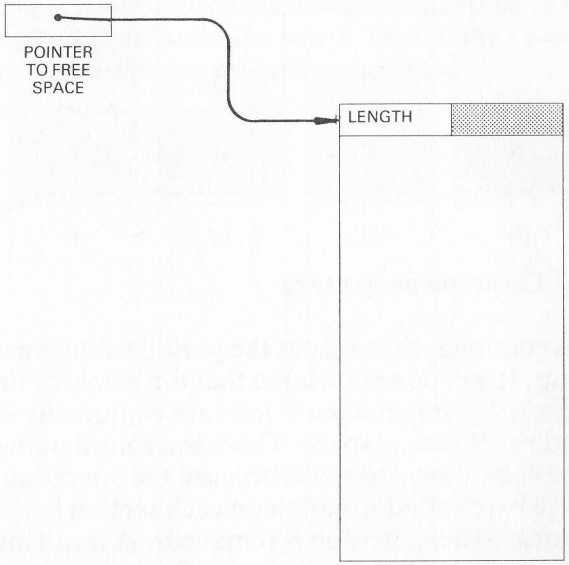


Figure 5.2a – New user heap (empty)

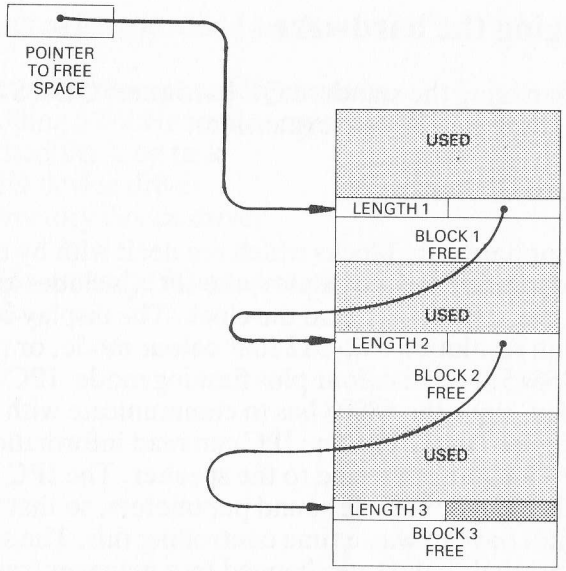


Figure 5.2b – User heap (partially used)

Figure 5.2 illustrates how a user heap can be operated, with the help of MT.ALLOC (trap #1, D0=$0C) and MT.LNKFR (trap #1, D0=$0D). The user heap is set up intially by linking an area of RAM into a non-existent heap (with free space pointer = 0). This is shown in fig 5.2a.

The Job allocates spaces for itself within this user heap, all spaces being linked together in a linked list format. As memory is allocated and freed, the user heap will fragment. At some stage in the future, there will be many free blocks within the heap. In order to find which one to use next, the utility routines will search down the list to find a suitably sized gap, which is then allocated. This is shown in fig 5.2b. Note that the user heap can be extended into any other area of memory (Job workspace of common heap) simply by linking it onto the end of the free space list.

5.3 Managing the hardware

As well as managing the standard QL hardware, QDOS has been designed to allow easy system expansion.

5.3.1 Standard QL hardware

There are four hardware blocks which are dealt with by the manager traps. They are the display, the IPC (includes keyboard and sound), the serial ports, and the clock. The display control select the high resolution 512*512 four colour mode, or the lower resolution 256*512 eight colour plus flashing mode. IPC control is more complex, since the 68008 has to communicate with the 8049 second processor (the IPC). The IPC can read information from the keyboard and output sound to the speaker. The IPC looks after all of the programmable sound parameters, so that the 68008 does not have to waste time controlling this. The serial ports can have their baud rate changed by a manager trap. The clock can be read, set or adjusted using manager traps $13, $14 and $15.

5.3.2 QDOS Extensions

The OS can be extended by adding routines to service interrupts and device drivers.Such additional routines will usually (although not necessarily) be associated with hardware which has been added to a standard QL. All interceptions are carried out through the use of linked lists. A linked list consists of a series of pointers. The first routine on the list is entered. Upon return from that routine, control passes to the next routine on the list, and so on until all routines have been called. Clearly, the nearer to the front of a list that a routine is, the higher its priority is, because it could choose not to pass on to the next routine. Since system calls are dealt with last, it is possible to replace system routines with new ones tailored to specific applications, as required.

The five linked lists provided are for:

external interrupt service routines
a polling 50/60Hz service routine
a scheduler loop task
an IO device driver
a directory device driver

More details are to be found in section 5.4 under manager traps $1A to $23.

5.4 Manager traps – Reference section

This section provides all of the information about the manager traps. Manager traps are all accessed via trap #1, the particular function being selected by the value in register D0. The following summary table lists all of the manager traps in numeric order. For a complete description of each, together with explanation and requirements of the calling code, you should refer to the later sections of this chapter.

Manager Trap summary

D0	Name	Description
00	MT.INF	Provide current job and system information
01	MT.CJOB	Creates a Job in transient program area
02	MT.JINF	Provide information on a Job
04	MT.RJOB	Removes a Job from transient program area
05	MT.FRJOB	Force remove Job in transient program area
06	MT.FREE	Finds largest free transient program space
07	MT.TRAPV	Sets per-Job pointer to trap vectors
08	MT.SUSJB	Suspends a Job
09	MT.RELJB	Releases a Job
0A	MT.ACTIV	Activates a Job
0B	MT.PRIOR	Changes a Job's priority
0C	MT.ALLOC	Allocates an area in a heap
0D	MT.LNKFR	Links a free space (back) into a heap
0E	MT.ALRES	Allocates resident procedure area
0F	MT.RERES	Releases resident procedure area
10	MT.DMODE	Sets or reads the display mode
11	MT.IPCOM	Sends a command to the IPC
12	MT.BAUD	Sets the baud rate
13	MT.RCLCK	Reads the clock
14	MT.SCLCK	Sets the clock
15	MT.ACLCK	Adjusts the clock
16	MT.ALBAS	Allocates Basic program area
17	MT.REBAS	Releases Basic program area

18	MT.ALCHP	Allocates common heap area
19	MT.RECHP	Releases common heap area
1A	MT.LXINT	Links external interrupt routine into QDOS
1B	MT.RXINT	Removes an external interrupt routine
1C	MT.LPOLL	Links a polling 50/60Hz routine into QDOS
1D	MT.RPOLL	Removes a polling 50/60Hz service routine
1E	MT.LSCHD	Links a scheduler loop task into QDOS
1F	MT.RSCHD	Removes a scheduler loop task
20	MT.LIOD	Links an IO device driver into QDOS
21	MT.RIOD	Removes an IO device driver
22	MT.LDD	Links a directory device driver into QDOS
23	MT.RDD	Removes a directory device driver

MT.INF TRAP#1 D0=0

Get 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 variables
A1	A1	preserved
A2	A2	preserved
A3	A3	preserved

Error returns:

none

Description:

This trap returns information about the current state of the system. The ID code for the current Job, and it's version number are returned. The pointer to the base of the system variables is also returned.

5.4.1 Creating and deleting Jobs

A Job is created in the transient program area of the QL's memory. It is essential that the memory allocated to the Job is sufficient to hold any executable code, a stack and working storage. If a Job is activated, execution will commence at the base address of the Job unless a different starting address is given. The start address is always given as an *absolute* address. It is not necessary for a Job to have any executable code. The Job could just be set up as a storage area for another Job. However, if the Job doesn't have any executable code, it must never be *activated*. Such an activation would almost certainly lead to a total system crash.

The Job area is essentially split into two parts. The code is defined to start from the base of the Job. All remaining space above the code, but below the top of the Job area is data space. When a Job is activated for the first time, (A6) points to the base of the area, (A6,A4) points to the bottom of the data space, and (A6,A5) points to the top of the area. If the Job is in a standard format, there may also be some information on the stack when it is activated. A word which contains the number of channels opened for the Job by the creating Job is placed at the top of the stack. This is followed by the appropriate number of channel IDs (long words), for the basic input, output and reporting channels respectively. These are followed by the Job's *command* string in standard format.

MT.CJOB TRAP#1 D0=1

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

Description:

This Job creation trap has two effects. It allocates space for the Job in the transient program area, and sets up a Job entry in the scheduler tables. However, this in itself doesn't cause the Job to be fully initialised. The only initialisation which occurs is that two words of 0 are put on the stack. The parent Job would normally load the new Job into the allocated area of memory after this system call. The stack pointer (in the Job control area – see later in this section) is initially set to point to the two zero words on the stack. These are placed at the highest addresses in the Job's data area.

If the Job is to have some channels opened for it, or if a command string is to be passed to it, this can be done before the Job is activated. The owner Job ID passed in D1 should be zero if the Job is to be independent. If the current Job is to become the new Job's owner, then D1 should be passed as a negative value. In operating system versions 1.03 and earlier, A1 should always be set to zero on entry.

MT.JINF TRAP#1 D0=2

Get 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 of Job
D3		D3	MSB negative if suspended LSB is the Job's priority
A0		A0	base address of Job
A1		A1	undefined
A2		A2	preserved
A3		A3	preserved

Error returns:

NJ Job does not exist

Description:

This trap returns the current status of a Job. It can be used to check the status of a complete tree of Jobs. If it is to be used in this way, D2 should always contain the ID of the Job at the top of the tree. In order to scan a complete tree, the trap is made with D1 containing the return value of the previous call. When the end branch of the tree has been reached, D1 is returned equal to zero.

MT.RJOB TRAP#1 D0=4

Remove Job from transient program area

Call parameters		Return parameters	
D1.L	Job ID	D1	undefined
D2		D2	undefined
D3.L	Error code	D3	undefined
A0		A0	undefined
A1		A1	undefined
A2		A2	undefined
A3		A3	undefined

Error returns:

NJ Job does not exist
NC Job not inactive

Description:

This trap removes a Job and all of its subsidiaries from the transient program area. This trap will only remove inactive Jobs. Active Jobs can be force removed using MT.FRJOB D0=5.

Note that this trap is not guaranteed atomic and cannot remove Job 0.

MT.FRJOB TRAP#1 D0=5

Force remove Job from transient program area

Call parameters		Return parameters	
D1.L	Job ID	D1	undefined
D2		D2	undefined
D3.L	Error code	D3	undefined
A0		A0	undefined
A1		A1	undefined
A2		A2	undefined
A3		A3	undefined

Error returns:

NJ Job does not exist

Description:

This trap inactivates a complete Job tree, and deletes all Jobs within the tree. If D1 is a negative word, then the deleted Job is the current one. If there is a Job waiting for the completion of any Job which is removed, that Job is released with D0 set to the error code (see MT.ACTIV D0=A).

Note that this trap is not guaranteed atomic and cannot remove Job 0.

MT.FREE TRAP#1 D0=6

Find largest contiguous free transient program space

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

Error returns:

none

Description:

This trap simply finds the largest contiguous free space which can be allocated in the transient program area. The length of the space is returned in D1.

MT.TRAPV TRAP#1 D0=7

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	undefined
A2		A2	preserved
A3		A3	preserved

Error returns:

none

Description:

It is possible for a Job to redirect traps and exception vectors which are not used by QDOS. This redirection is defined by a table (see below). If a Job sets up a table of trap vectors for itself, that table will automatically be used whenever that Job is being executed. A Job will initially use the table of the Job which set it up, even if it isn't owned by that Job. This state of affairs persists until the new Job sets up its own table. Vector tables used by other Jobs are not affected. If the Job ID is a negative word, the table will be set up for the calling Job.

The table consists of a long word address for each trap or exception, in the following order:

table start	$00	address error
	$04	illegal instruction
	$08	zero divide
	$0C	CHK (check register against bounds)
	$10	TRAPV (trap on overflow)
	$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

5.4.2 Job Control

Jobs always exist in one of three well defined states. They are either *active* (sharing CPU resources with other Jobs), *suspended* (waiting for IO or another Job) or *inactive* (occupying memory but unable to use CPU resources). The only practical difference between an inactive Job and one which has been suspended indefinitely is that the latter cannot be removed by MT.RJOB D0=4.

The information about a Job, its current status, priority, owner, size etc. is contained in the Job control area. The Job control area is in the transient program region of memory and immediately precedes the area allocated to the Job. In QDOS V1.03, the base of the Job control area is $68 bytes before the Job itself. This is liable to change in future versions. For QDOS V1.03, the memory organisation is:

addr.	length	name	description
$00	long	LEN*	total length of Job control + Job area
$04	long	START	start address on activation
$08	long	OWNER?	Job ID of this Job's owner
$0C	long	HOLD?	pointer to a byte which will be cleared when the scheduler releases the Job (see MT.SUSJB D0=8)
$10	word	TAG*	tag for Job as allocated by MT.CJOB

$12	byte	PRIOR?	current accumulated priority. This increments when a Job is active but not executing. It is set to zero when the Job is executing. The scheduler allows the Job with the highest accumulated priority to execute.
$13	byte	PRINC?	this priority increment is the initial priority of the Job. Basic activates Jobs at priority $20.
$14	word	STAT⋆	Job status: 0 is not suspended >0 is number of frames before release −1 is suspended (IO or MT.SUSJB) −2 is waiting for a Job to finish
$16	byte	RELA6?	MSB is set if next trap #2 or #3 has relative addressing (as set by trap #4)
$17	byte	WFLAG?	set if a Job is waiting for this one
$18	long	WJOB?	ID of Job waiting for this to finish
$1C	long	TRAPV?	pointer to trap redirection vectors
$20	8 long words		saved values of D0 to D7
$40	8 long words		saved values of A0 to A7
$60	word		saved value of status register
$62	long words		saved value of program counter

⋆ means value should not be changed
? means value may be changed by a trap, or directly (but with care!)

MT.SUSJB TRAP#1 D0=8

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 control area
A1	address of flag byte	A1	preserved
A2		A2	preserved
A3		A3	preserved

Error returns:

NJ not a valid Job ID

Description:

This trap suspends a Job. The suspension period may be indefinite, or for a given amount of time. The timeout period is defined as a number of frame periods, so suspension may be for up to 32K frame periods (about 10 minutes). If the timeout period is specified as −1, the suspension will be indefinite. No other negative value should be used. If the Job ID is a negative word, the current Job will be suspended. The flag byte will be cleared when the Job is released. If there is no flag byte, then A1 should be set to zero. If the Job was already suspended, the suspension will be reset. All Jobs are rescheduled.

Note that this trap is not fully atomic because it invokes the scheduler!

MT.RELJB TRAP#1 D0=9

Releases 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 control area
A1		A1	preserved
A2		A2	preserved
A3		A3	preserved

Error returns:

NJ not a valid Job

Description:

This trap causes a Job to be released from a suspended state. All Jobs are rescheduled after this call.

Note that this trap is not fully atomic because it invokes the scheduler!

MT.ACTIV TRAP#1 D0=A

Activates 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 (−1 or 0)	D3	preserved
A0		A0	base of Job control area
A1		A1	preserved
A2		A2	preserved
A3		A3	preserved

Error returns:

NJ Job does not exist
NC Job already active

Description:

This trap causes the defined Job in the transient program area to be activated. Execution commences from the start address which was defined when the Job was created. The activity of a Job can also be controlled by adjusting the Job's priority level (MT.PRIOR D0=B). A Job becomes inactive if its priority is set to zero.

Execution of the current Job will continue if the timeout is set to zero, otherwise the current Job will be suspended until the activated Job has finished. This trap will then return with the error code from that Job.

Note that this trap is not fully atomic because it invokes the scheduler!

MT.PRIOR TRAP#1 D0=B

Changes 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 control area
A1		A1	preserved
A2		A2	preserved
A3		A3	preserved

Error returns:

NJ Job does not exist

Description:

This trap changes the priority of a Job. If D1 is a negative word, it will change the priority of the current Job. The Job will be inactivated if its priority is set to 0. This Job re-enters the scheduler, so if a Job sets its own priority to 0, it will be inactivated instantly. This call is not fully atomic because it invokes the scheduler.

5.4.3 User Heap Management (see also chapter 8)

Under certain circumstances, user heap management may need to be atomic. Two traps have been provided to take account of this.

User heaps are allocated in multiples of 8 bytes. Free space is linked together by using two long words per space. The first long word contains the length of the space. The second long word is the relative pointer to the next free space. User heaps are totally relocatable, because all pointers are relative. The entire allocated area can be used by the user code provided that the code is programmed to remember the length of the items in the heap. When the area is allocated, the first long word contains the length of the area, so that it can be retained by the user code if desired.

The user code must keep one pointer to the free space in the heap. An explanation of this is provided in section 5.2.5. The pointer should be a relative pointer to the free space in the heap (a long word). If the heap doesn't have any free space, either because it is full or doesn't exist, this pointer will be zero.

Heaps are set up by linking an area of RAM into a non-existent heap (with free space pointer = 0). Heaps can be expanded by linking in an area of RAM. This should optimally be contiguous with the current top of the heap.

MT.ALLOC TRAP#1 D0=C

Allocates an area in a heap

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

Error returns:

OM no free space large enough

Description:

This trap allocates an area in a heap for use by user code. The trap is fully atomic. A6 is used as a base address for this trap, so A0 must be relative to A6.

MT.LNKFR TRAP#1 D0=D

Links a free space (back) into a heap

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

Error returns:

none

Description:

This trap links a free space of memory (back) into a heap. A6 is used as a base address for this call, so A0 and A1 are both relative to A6.

MT.ALRES TRAP#1 D0=E

Allocate resident procedure area

Call parameters		Return parameters	
D1.L	no. of bytes required	D1	undefined
D2		D2	undefined
D3		D3	undefined
A0		A0	base address of area
A1		A1	undefined
A2		A2	undefined
A3		A3	undefined

Error returns:

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

Description:

The resident procedure area sits at the top of memory, and is normally allocated at power up. The reason for this is that all transient procedures sit below the resident procedure area in memory. If the resident procedure space expanded when there were transient programs present, it would be necessary to relocate the transient programs. They are often not relocatable, so this is impossible. However, in the exceptional case of there being no transient procedures in the machine, it is possible to both extend and contract the resident procedure area. The two traps MT.ALRES D0=E and MT.RERES D0=F are incorporated for this task. The allocation is for a number of bytes. Note that the Basic function RESPR(n) can be used to allocate n bytes of resident procedure memory.

MT.RERES TRAP#1 D0=F

Release resident procedure area

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

Error returns:

NC unable to release (TRNSP area not empty)

Description:

This trap releases resident procedure memory for use by the system. It can only be used when the transient program area is completely empty. See also MT.ALRES D0=E for the allocation of resident procedure memory.