# COMPUTER ONE MONITOR

## for the Sinclair QL computer

## A USER GUIDE

Computer One welcomes ideas and comments. These and any bug reports or further enquiries should be sent on the report form at the back of this manual to:

Technical Enquiries,
Computer One Ltd.,
Science Park,
Milton Road,
Cambridge CB4 4BH.



# TABLE OF CONTENTS

## TABLE OF CONTENTS (continued)



# INTRODUCTION

The Computer One Monitor is an invaluable tool for anyone developing software for the QL. It has been designed to take full advantage of the QDOS operating system, and can handle several jobs at once.

It has a large and powerful set of instructions, most of which can be invoked with only one or two key depressions. Users can dump and disassemble memory, trace programs, and control jobs. Memory can be modified, moved and searched. When tracing you can set breakpoints and watch for particular events. There is also an extensive set of channel management commands to allow user definition of screen layout, and output to any QL device, for example, a printer or file. The sophisticated user will be able to take advantage of the Monitor's ability to clone itself to monitor several jobs at once.

This user guide assumes that the user has a knowledge of 68000 assembly language, a basic knowledge of QDOS, the QL operating system, including the use of TRAPs to make system calls and the concept of QDOS jobs.

## USING THIS GUIDE

This section contains a brief overview of each chapter.

CHAPTER 1 describes how to backup the monitor cartridge, gives a general overview of the monitor and explains how to start the monitor.

CHAPTER 2 describes how to use the monitor, explaining the syntax of monitor commands by working through a short debugging session using one of the example programs on the system microdrive cartridge.

CHAPTER 3 describes each command in detail, giving examples of the use of each command.

CHAPTER 4 contains a summary of the commands, giving the full syntax of each command.

# CHAPTER ONE
# GETTING STARTED

## 1.1 BACKING UP

The supplied microdrive cartridge should be backed up immediately on receipt. This cartridge should be treated as a Master copy. It is recommended that you make two backup copies using the Master cartridge only as an emergency backup and not to run the software. Backing up may be done by running the supplied 'CLONE' program as follows:

1. Place the Master copy in microdrive 2 (the right hand side drive).

2. Place a blank cartridge in microdrive 1.

3. Enter the following command:

```
LRUN mdv2_clone <ENTER>
```

4. The QL will respond with various instructions to name the new cartridge and initiate the copying, MAKE SURE THE MASTER IS IN DRIVE 2.

5. The 'cloned' system may be used as soon as the microdrives have stopped running.

Repeat the procedure with another cartridge, and store the master and one of the copies in a safe place. Use the remaining copy as your working master — only use the others in an emergency. Please note that you may only copy the software for your own use.

## 1.2 OVERVIEW

The Monitor is designed to give you an understandable view of the machine's current state. It does so mostly by giving you dumps of memory in various forms (dump for data, disassembly for programs), plus a copy of the machine's internal registers, i.e. the data registers D0 to D7, address registers A0 to A7, the program counter PC, system stack pointer SS, and status register SR.



The data/program/register displays are kept in separate windows for clarity, and the form of each can be altered to suit your own taste. To keep things simple, a small region in the machine is considered as the "main" area under concern, and its addresses are simpler to specify and normally appear as "program counter" relative.

In actual fact, since the program counter tends to change during tracing etc., another pseudo-register is used to keep the bottom address relative to which common addresses are taken — it is called BP (base pointer). The top of the area is held in a pseudo-register called TP (top pointer). In addition there are 8 other pseudo registers called R0 to R7 which you may use for your own purposes, they are useful for storing temporary results, especially addresses, for subsequent use.

Under normal circumstances you will probably be interested in debugging a single program, and it is this which will be held between BP and TP. The program will need some stack-space, and also has to be loaded into the machine into a "safe" area of memory. This is where the concept of a "job" is useful. The operating system on the QL (QDOS), can allocate areas of memory which will not conflict with other programs or data. Sections of such memory are often referred to as jobs, and can be referenced via a special number or "job_id". The Monitor's load command is responsible for asking QDOS to allocate an area of memory, as a job, for your program (or data). Once loaded BP, TP, A7 (the stack pointer), PC etc. are set to sensible values to allow you to start analysing your program and the memory which it affects.

The monitor can run several jobs simultaneously. Only the "current" monitor job would normally be traced, but any other job loaded and started from the monitor will be suspended by the monitor whenever a breakpoint or program error in that job occurs. The monitor can then be used to examine the job, including the registers which are set to their values when the error occurred.

## 1.3 LOADING AND RUNNING

In this and all subsequent sections it will be assumed that the program is in microdrive 1. It could equally well be stored on any device, e.g. floppy disks, and may be referred to as, say, flp1_c1mon, in such cases.

The Monitor runs as a QDOS job and is initiated by the superBASIC command:

```
exec mdv1_c1mon <ENTER>
```

When the monitor has been loaded a number of windows appear on the screen and you may have to press CTRL-C before you start entering monitor commands. To re-enter superBASIC or other jobs which have an active prompting cursor, press CTRL-C again. The monitor can take its input from any channel; usually this will be from the keyboard, but you may wish to execute a particular set of commands every time you start the monitor. When the monitor has just been EXECed, you can use the down arrow or up arrow keys to open the files 'mdv1_boot_mon' (down arrow) or 'flp1_boot_mon' (up arrow) and the monitor will take its initial commands from one of these files. See section 3.6 for more information on channel management.

A typical session might start and end as follows:

```
exec mdv1_c1mon          (start Monitor)
(CTRL-C)                 (if necessary to switch
                         from BASIC to Monitor)
load mdv2_myprog         (get program to be debugged
                         into memory)
dis                      (disassemble from start)
:                        (etc)
                         (debugging your program here)
:
kill 20003               (get rid of job (if necessary))
quit                     (exit monitor)
```

When you EXEC the monitor, there is no current job to be monitored, so the monitor is set to look at SuperBASIC.

Remember when using the monitor that you are working with the machine at a very low level and that it can be easy to crash the QL unless you are very careful, particularly when poking values into memory, or moving blocks of memory in the machine.



# CHAPTER TWO
# USING THE MONITOR

This chapter describes how to start using the Computer One monitor, giving a description of the general format of commands, the types of parameters accepted by commands and some examples of the use of the more common commands.

## 2.1 COMMAND FORMAT

The general format of a command is:

Command [#channel[,]] [‹parameters›] [[;] flags]

where "[ ]" is used to denote optional items, Some of the commands will demand that their parameters are written explicitly, but most use a "default" to save you having to enter them. All commands can be abbreviated. To obtain a list of the commands available type:

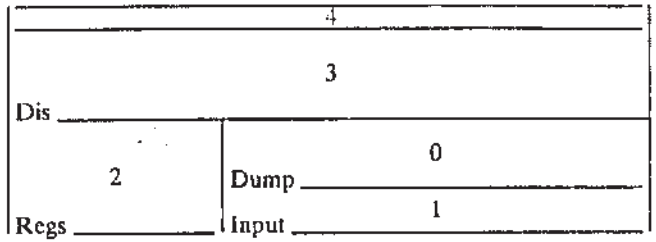
```
help (ENTER) (or simply h (ENTER))
```

in response to the C1MON› prompt. (All command lines need ENTER at the end, and this will be omitted from now on.) More than one command can be entered on a line by using a colon as a command separator.

If two commands start with similar sets of letters then the first command in this list will be chosen. (For clarity commands are always written in full in this manual.)

## 2.2 CHANNELS

When the monitor is loaded a screen appears in the following format:




The screen has five channels open, numbered 0 to 4. Channel 1 is the input channel and it is in this window that the monitor command prompt appears. All commands have a default channel which is used if no channel is specified. Most commands use channel 0 as their default. The concept of channels is very similar to that in SuperBASIC and new channels can be opened to allow disassembly, memory dumping etc. to other devices, such as printers or microdrive files. More detailed information on channel management is given in chapter 3, section 3.6.

## 2.3 PARAMETERS

Most of the commands have some parameters following the channel number. These parameters, depending on the command, can be colours (for channel management), line counts, values, addresses, ranges of addresses, or job or channel ids.

**Colours**
A colour is specified as in SuperBASIC and can include values to produce stipples. It is easier to specifiy stipples using binary or octal values.

**Line counts**
A line specifier is a decimal integer which specifies the number of lines to be output to the screen in a dump or disassembly command.

**Job and channel ids**
A job-id is a 32-bit value, which identifies a monitor job, for example 20003. A channel id has a similar format and identifies a channel. Note that if the id starts with a hex letter it must be preceded by a zero (or $).



### 2.3.1. Value Specifications

Values can be specificaed as numbers (any base) or combinations of numbers using the operators +, –, *, /, !, &, <, >, ^ and %. A value can also be an ASCII string, the contents of an address or the value of an address. There is a special null operator (.) which can be used to resolve ambiguities. See section 2.3.3.

The syntax for a value can be formally specified as follows (a vertical bar indicates alternative selection):

```
<value> =    [$]<hex number> | &<decimal number> |
             ^<octal number> | %<binary number> |
             "<string>" | '<string>' | ~ <value> |
             +<value> | -<value> | <value>+<value> |
             <value>-<value> | <value>*<value> |
             <value>/<value> | <value>!<value> |
             <value>&<value> | <value><<value> |
             <value>><value> | <value>^<value> |
             <base n>%<n-ary number> |
             ?<address> | @<address>
```

The operators above which are not self-explanatory are

! (bit or) & (bit and) < (shift left)
> (shift right) ~ (not) ^ (mod)
% (allows the given value to be specified in the given base)

A number defaults to the hex value if it is not preceded by a $, &, % or ^. Hex numbers must start with a digit or a $ (i.e. put a 0 or a $ in front of the number if it starts with a letter). The precedence of operators is essentially left to right (monadic operators have higher precedence than diadic), but can be altered by surrounding values in square brackets. Note that if a string is used in an arithmetic expression its value is considered to be the long word made up of the ASCII codes of the rightmost four characters of the string.

The operator '?' in front of an address returns the long word at the specified address. The operator '@' is used to convert an address type into a value type.

Example : Using indirect addressing

```
@25(a0) - address found by adding $25 to contents of A0.
```

This could be written 25 + ?A0, i.e. the value $25 added to the contents of A0.

If all this seems a bit confusing the best way to find out about values is to use the eval command which displays the 32-bit hex form of a value. Below are some examples, but we suggest that you try more examples yourself using the monitor.

Examples:

```
eval 0f+&20      = $00000023   (15 + 20 in hex)
eval 8+[4*3]     = $00000014
eval 'a'         = $00000061
eval 'dcba'      = $64636261
eval 81-%10      = $0000007f
eval ?BP         = contents of base pointer
eval @(a6)       = address in a6 as a value
eval ?a6         = value in a6 (same as above value)
eval 8 ^6        = $00000002   (8 mod 6)
eval ^44         = $00000024   (44 in base 8)
```

### 2.3.2 Address Specification

A lot of commands need an address as one or more of their parameters. Normally addresses specified are taken relative to an internal register called BP. However absolute addresses can be specified by following the relevant address with a £ sign. Registers can also be considered as addresses. Think of a0 as being were the value of the register a0 is held. In addition addresses like (a0) 6(a2) 22(a1,d2) or even 12(28064) may be given, with the usual assembly language interpretation. (12(28064) means $12+longword contents of 28064.)

The formal specification for an address is as follows (curly brackets followed by * indicate that the part in brackets may be repeated zero or more times).

```
<address> = <value> | <value>£ | register |
    [<value>] ( <address>[.w | .l] {,<address>[.w | .l]}* )

<register>= d0...d7 | a0...a7 | r0...r7 | pc | sr | bp | tp
```

The last part of the address specification may look rather imposing but is just a formal specification of the indirect addressing modes. Examples of allowable indirect addresses are:

```
(a0)          — contents of register a0
(3f000)       — contents of location 3f000
10(a0,d1.w)   — $10 plus contents of a0 + sign extended contents of d1
(a0,d1,d2)    — contents of a0 + contents of d1 + contents of d2
```

Where a command takes an address range this is specified as

```
<addr range> = <address1> | <address1>,<address2> | = |
                 <address1>\<value>
```

This means that an address range is as follows



1. a single address
2. all the addresses in the range ‹address1› to ‹address2›−1
3. if "=" is specified then the range used will be from the base of the current job to the top of the current job, i.e. (BP), (TP)
4. if a ‹value› is specified then the command using the range will operate on ‹value› addresses starting at the given address.

Examples:

```
&100        [ the single address 100(BP) ]
0d0,0f0     [ addresses in the range 0d0(BP) to 0f0(BP)-1 ]
d0\20       [ all the data registers = 8 longwords ]
d0,a0       [ same as above, since address registers follow data
              registers in job header ]
```

The registers r0. .r7 are a set of registers which can be used for any purpose. For example you may wish to temporarily store all the data registers in the r registers.

Again the best way to find out about addresses is to practise using them. You can use the eval command with the @ operator to get addresses and convert them to value types.

Examples:

```
eval @30              = $30 + contents of BP
eval @30£             = $30
eval @25(a6,d0.w)     = $25 + contents of d0 + sign extended contents of d0
eval @BP              = address where BP is held
```

### 2.3.3 Ambiguities in parameters

A large number of the monitor commands can take default values and this can lead to ambiguities when specifying parameters of these commands with operators which are both monadic and diadic. For example the command

```
poke     4 -3
```

can be taken to mean 'poke the address 4(bp) with the value -3' or 'poke the address 1 (bp) with the default value'. The monitor syntax checking would take the second meaning. To resolve ambiguous cases you can use the null operator '.' to force the diadic operator to be taken as monadic, i.e. use

```
poke     4 .-3
```

to force the first meaning. There are very few cases where ambiguities are likely to arise, but this operator will allow you to force the required meaning.

## 2.4 FLAGS

The flags control the precise action of certain commands, and are mostly defaulted. They can be set globally using the flags command, but can be overridden by specifying the flag with a specific command. Each flag can be preceded by 'no' to turn it off.

| Flag | Command(s) | Action | Default |
|---|---|---|---|
| rel | dis, dump, reg, jobs | Output addresses in BP relative format | On |
| pc | dis, dump, regs | Output addresses for lines displayed. For reg displays PC, status register and BP | On |
| hex | dis, dump | display hex code or hex memory dump. | On |
| asc | dump | display ascii memory dump | On |
| an | regs | display address registers | On |
| dn | regs | display data registers | On |
| rn | regs | display user registers | Off |
| dis | trace | display disassembled code while tracing | On |
| regs | trace, job, load | display register dump after command | On |
| trap | trace | trace through traps | Off |
| jsr | trace | trace through jsr/bsr subroutines | On |
| break | trace, go | set breakpoints before executing instruction | On |
| watch | trace | set watchpoints before executing instruction | On |
| job | commands which output addresses | displays address in form xxxx+job, not xxxx(bp). Where "job" is base of current job. (See section 3.1.1 for use of flag) | Off |



Examples:

| | |
|---|---|
| `dump 28000 ;nohex` | Memory dump in ASCII form only. |
| `regs #3 ;rn nopc` | Display registers including user registers, no PC or BP register. |
| `trace ;notrap nojsr` | Treat execution of traps and jsrs as single instructions. |

## 2.5 EXAMPLE PROGRAM

This section gives an informal introduction to the basic commands of the monitor. Chapter three gives a detailed description of all the commands.

The system microdrive cartridge has two files — example_asm and example_cde. We shall use the _cde file as an example program. It increments register d0 50000 times and then makes a call to one of the QDOS traps to get the version of the operating ststem. The program has an error in it, which we can fix while monitoring it.

In this example it is assumed that the monitor cartridge is in microdrive one.

First start up the monitor

```
EXEC mdv1_c1mon
```

and press CTRL-C to get the flashing cursor in the monitor window with C1MON› prompt. To load the program into the monitor use the **load** command:

```
load mdv1_example_cde
```

When the monitor has loaded this file it will have set up a job for it, but the job will not have started to execute. If you now type in the command

```
jobs
```

with no parameters, information about the current jobs on the QL is displayed in window 0. There should now be two jobs, assuming there were no other jobs running when you started the monitor; the job with job-id 0 is SuperBASIC, the other job is the one you have just loaded.

To look at the first few disassembled instructions of the job we use the **dis** command, which has the form:

```
dis [#channel[,]][address][,lines][[;]]flags]
```

The default disassembly address is the one after the last address examined, or the start of the current job, if it has not yet been disassembled. The default number of lines to display, i.e. instructions to disassemble, is 8, since the disassembly window is eight characters high. So we can now enter

```
dis      or
dis 0,8
```

and the first eight instructions are disassembled in window 3. The display shows the address, the hex code and the disassembled code for each instruction. The output of the addresses and the hex code can be turned off using the nope and nohex flags. Remember that if you are trying to disassemble an area that contains data rather than code, the output will be meaningless. Areas of data can be examined using the **dump** command, which has the same parameters as the **dis** command. Each line of dumped memory consists of eight bytes giving the hex values, followed by eight bytes giving the equivalent ASCII characters. Non-printable ASCII characters are displayed as dots. The nohex and noasc flags can be used to turn off part of the display, for example

```
dump 0,8 noasc
```

When you are using the **dump**, **dis** or **trace** commands you can use the down arrow to repeat the last command from the last used address. Thus to disassemble a large number of instructions from the start of the program use

```
dis 0,8
```

followed by repeated presses of the down arrow. The up arrow key can be used to disassemble the few instructions before the current one or to dump the piece of memory just before the current address. With the **trace** command the up arrow key will skip the next instruction, i.e. it will not be executed.

We can now start tracing our job using the **trace** command. This command has a simliar format to dump and dis:

```
trace [#channel[,]][address[],lines][[;]flags]
```

the default lines for this command is one, allowing single step tracing easily. Enter the command

```
trace        or
trace 0,1
```

The first instruction has now been executed and the disassembled code of the next one has been displayed. Note that a dump of the registers has automatically been done. You can now use the down arrow to trace through single instructions. The first few instructions of this program make up a loop which exits when d1 reaches 50000 and since we don't want to single step through this



we can set a break point at the first instruction past the end of the loop. The go command can then be used to execute until a break point or error conditions is encountered. The first instruction past the loop is $0C(bp), so set a break point at this address:

```
break 0c
```

If you now type

```
breaks
```

all current break points are displayed in window 0. Now type

```
go
```

The program will stop running when the breakpoint is reached. If you now issue the regs command all registers are displayed. You can see that d1 now has the value 50000 (=$C350). Although we won't trace this piece of code again we can remove the break point by typing

```
nobreak 0c
```

If you disassemble the piece of code from the break point by typing

```
dis 0c
```

you can see that the code sets up some registers and uses a trap call. This trap returns the version of the QDOS operating system in d1. We now want to execute the next instruction. To do this we can enter

```
trace 0c,1
```

or just

```
trace
```

since the PC is at the correct instruction. If you now examine the registers you will see that d0 has the value 1. The value should be 0 if the trap is to work correctly. To change the value of a register, or indeed of any memory location, you can use the **poke** or **.** command. Both commands are the same, but poke uses a default length of a byte, . uses a default length of a long word and redisplays the registers. The formats of the command are:

```
poke[_<size>] <addr range> <value>

.[_<size>] <addr range> <value>
```

‹size› = b or w or l. The command fills all bytes, words or longwords in the specified address range with the given value.

Since we want to poke d0.l we shall use . :

```
.d0 0
```

We can now continue tracing. The next instruction to be traced is the trap. When tracing, a trap is usually treated by the monitor as a single instruction, so that you don't have to trace through the whole trap. This means that after tracing the trap instruction the next one will be the one after the trap in the user code. However, if you do want to trace through traps you can use the trap flag when issuing the **trace** command. The same applies to subroutines called using jsr or bsr. The jsr flag can be used to trace through subroutines. (The default for subroutine tracing is on.)

We shall assume that we are not tracing the trap, so just press the down arrow or use the **trace** command to continue tracing. Register d1 should now contain the four ascii characters giving the operating system version, for example '1.02'.

If you now press the down arrow key twice more the last two instructions of the program will put the value in d2 into the area of memory just beyond the end of the code. To examine this area of memory use the **dump** command

```
dump (a0),1
```

The first 4 bytes displayed will give the operating system version number.

Having traced through this small program we can now kill the job using the monitor **kill** command:

```
kill <job-id>
```

The ‹job-id› can be obtained by using the **jobs** command. The current job is the one whose base is given in BP relative form. Another way of killing the job is to have code in the program to explicitly kill a job. You would normally have code to do this if the program you are developing is eventually going to be a QDOS job.

Although this is a fairly trivial example, it helps to illustrate some of the more commonly used commands in the monitor. Chapter three contains a detailed description of these commands and all the other monitor commands, including memory searching, memory moving, watching for particular events, job control commands and channel management commands.



# CHAPTER THREE
# THE MONITOR COMMANDS

This chapter gives the full dscription of each of the monitor commands, giving the full syntax, the default output channel, the relevant flags and the default values of parameters not entered. The flags cannot be abbreviated and are switched "off" by preceding the name with "no". All flags except trap, rn and job are "on" at system start-up. Note that the default channel for any command can be altered by issuing the command name followed by a channel number and full-stop. Several commands can be entered in a single line by separating them with a colon (:).

Example:

```
reg #0.        (set default register dump to channel 0)
```

## 3.1 EXAMINATION

### 3.1.1 Disassembling code

**Command:** `dis [#chan[,]] [<address>][,<lines>] [[;]flags]`

**Default Channel:** 3
**Default Address:** current PC in job header or address after last code disassembled
**Default Lines:** last ‹lines› specified or if ‹lines› is 0 the height of the current window (or 8 if output is not to a window).
**Flags:** {no]hex [no]pc [no]rel [no]job

Once this command has been issued the up-arrow and down-arrow keys (without being followed by ENTER) can be used to scan upwards and downwards through code rapidly.

**Action:** an area of memory, from the given or default address, is disassembled in standard Motorola format. Bad code is disassembled as far as possible, or ends with "?" if found to be meaningless. The number of instructions disassembled is given by the number of ‹lines›. It is advisable to use a ‹lines›

value which will allow all the disassembly to fit into a window. The output for each instruction shows the address in BP relative format (unless the rel flag is "off" or the address lies outside the current job, i.e. outside the range (BP), (TP)), the hex code for the instruction (unless the hex flag is "off") and the disassembled code. If the pc flag is "off" the address is not output. Note that instructions are always on word boundaries and that code disassembled may be initially incorrect if the address given is not at the start of an instruction.

Note that the job flag is useful if you want to produce output which can be reassembled. The job flag causes relative addresses to be output in the form

```
xxxx + job
```

where xxxx is the offset from the start of the job and "job" is a label which would have to be declared at the start of the file containing the code. To output the code to a file in this form you should open a file as one of the monitor channels (say #5) and disassemble to this file as follows:

```
dis #5, 0,<any number> ; nopc nohex job
```

Repeated presses of the down arrow key would then continue to output code to the given channel.

Examples:

```
dis 12        (disassemble window-height lines from 12(bp))
dis #0, 100£,6 ;nohex
              (disassemble 6 lines from absolute address 100 outputting to
              channel 0 with no hex code)
dis ((a7))    (disassemble code whose address is on top of the stack.
              Useful if the value on top of the stack is a return address.)
dis ??a7      (same as above example)
```

### 3.1.2 Displaying memory

**Command:** `dump [#chan[,]] [<address>][,<lines>] [[;]flags]`

**Default Channel:** 0
**Default Address:** 0 (BP) or address after last code dumped
**Default Lines:** last <lines> specified or if <lines> is 0 the height of the current window (or 8 if output is not to a window).
**Flags:** [no]hex [no]asc [no]rel [no]pc [no]job

Once this command has been issued the up-arrow and down-arrow keys (without being followed by ENTER) can be used to scan upwards and downwards through memory rapidly.



**Action:** an area of memory is dumped in the normally accepted manner (i.e. address, hex contents, ascii contents). If the rel flag is "off" the address is absolute; if the hex flag is "of" the hex dump is not output; if the asc flag is "off" the ascii dump is not output.

Examples:

```
dump 0,6      (dump 6 lines of output from address 0(BP))
dump 6(a0)    (dump window height lines from address 6
               + contents of register a0)
```

### 3.1.3 Displaying registers

**Command:** `regs [#chan[,]] [[;]flags]`

**Default channel:** 2
**Flags:** [no]an [no]dn [no]rn [no]pc [no]rel [no]job

**Action:** displays the current job's address and data registers, the status register, the program counter and job's base pointer (BP). These registers are used during tracing. If the pc flag is "off" then only the address and data registers are displayed. If the an or dn flags are "off" then the address or data registers are not output. If the rn flag is "on" then the monitor's set of user registers is displayed.

Examples:

```
regs          (output to default channel all registers except rn)
regs #0;rn nopc   (output a, d, r registers to channel 0)
```

## 3.2 MODIFICATION

### 3.2.1 Poking memory

**Command:** `poke[_<size>] <addr range> [<value>]`

**Default size:** b (bytes)
**Default value:** 0

**Note:** The size can be one of "b", "w" or "l", indicating byte, word or longword poking respectively. Note that using the status register SR causes the size to be set to "w" and using SR in forms of this command, other than to poke a constant value, may give unpredictable results.

**Action:** the memory specified (either by single address or a range) is poked to the value given. If a range of memory is to be set then the address increments by 1, 2 or 4 or the length of a string, if a string is supplied, until the second address is reached. Remember that poking memory must be done with great care.

Examples:

```
poke 81 46              (set location $81(bp) to $46 (byte value))
poke 200,300            (clear all locations from $200(bp) to $2FF(bp))
poke_w 20000,21000 23
                        (set all words from 20000 to 20FFE to $0023)
poke_l 8(a6) 1          (set the long word at address 8 + contents of
                        register a6 to $00000001)
poke 0\10 0             (set 16 bytes to 0 starting at 0(BP))
poke 20, 'hello'        (poke the string at $20(BP))
```

"." is an alternative to poke which is more likely to be used on registers.

**Command:** `.[_<size>] <addr range> [value]`

**Default size:** l (longword)
**Default value:** 0
**Flags:** [no]regs

**Action:** Same as poke, except that a register dump is done automatically at the end of the command. Note that this can be used to set a range of registers to the given value.

Examples:

```
.pc a0              (set PC register to start of current job)
.d6 3456            (set register d6 to $00003456)
.a2,a5 ?bp          (set a2, a3, a4 to the job base register)
.d0\20              (clear all data registers 8 longwords)
._w 6(a6,d2.w)      (clear word at address 6(a6,d2.w))
.0d0 23             (set longword at 0d0(BP) to $00000023)
```

### 3.2.2 Moving memory

**Command:** `move <addr range> <address>`

**Action:** copies a block of memory byte by byte from the range specified to the ‹address› specified.



Examples:

```
move 0£,8000£ 20000     (copy bottom $8000 bytes to 20000,
                         the screen)
move a0,a5 r0           (save registers a0. .a4 in r0. .r4)
```

## 3.3 VERIFICATION AND SEARCH

### 3.3.1 Comparing memory

**Command:** `compare [#chan[,]]<addr range> <address> [[;]flags]`

**Default channel:** 0
**Flags:** [no]asc [no]hex [no]pc [no]rel [no]job

**Action:** compares the range specified with the area starting at ‹address›. The first mismatch is reported (in a form similar to dump, giving what was found/ what it should be). After a move no mismatch should be found unless memory has altered since it was moved. If no mismatch is found the message 'ok' is displayed. Note that the specified flags apply to the dumped output.

Examples:

To check that the two moves above worked correctly use

```
compare 0£, 8000£ 20000
compare #3,a0,a5 r0
compare 0\20 100        (compare $20 bytes of memory from
                         (bp) with 256(bp))
```

### 3.3.2 Memory search

**Command:** `find[_<size>] [#chan[,]] <addr range> <value> [[;]flags]`

**Default size:** b (byte)
**Default channel:** 0

**Action:** searches the range for the given value, reporting using a dump, as in the **compare** command. If a string is given as the value then the comparison is case-independent, the step size for comparisons being one byte. The specified string may use the character '?' as a wild card to match with any other single character. Otherwise adding _b to the main command compares bytes located at every byte, _w compares words, with word spacing, and _l compares longwords, also with word spacing. Note that "=" can be used for the ‹addr range› to search through the whole of the current job.

Examples:

```
find 0£,40000 "c1mon"     (search all memory for C1MON string)
find_w 0,2000 4AFB        (look for "break" instruction from (bp) to
                           $2000(bp))
find_l = -1               (find, within the current job, $FFFFFFFF
                           word aligned)
find 0\&2000 30           (search 2000 decimal bytes starting at
                           0(bp) for $30)
```

## 3.4 MISCELLANEOUS COMMANDS

### 3.4.1 Loading jobs

**Command:** `load <device> [<address>]`

**Default address:** job base set up by monitor.

**Action:** loads the file named at the address given (if supplied). If no address is given then the file is loaded as a job. (This happens even if the file is only really a data file — the monitor will allocate a dataspace of $100 bytes for any code contained in the file.) The internal Monitor registers, bp (base pointer) and tp (top pointer), are set to the bottom and top of the job created.

This area will have addresses output in the form ????(bp), and, since bp is added to certain forms of address in any case, are the addresses which are most easily referenced (and probably the one most frequently used). The register set used is that of the new job's header (q.v. jobs), so that they reflect the values which would normally be set when the job is activated. The job's priority is set to zero, but the job is not suspended — if you wish to start the job, without tracing it, then use the go command, (N.B. do not start jobs which are only data!)

Examples:

```
load mdv1_myprog_exe      (load this file and set it up as a job)
load flp2_my_dat (r0)     (load this file into memory at the specified
                           address)
```

In this last example you should ensure that the memory at the address specified is available, probably by using the **heap** command and using the address returned as the load address. Loading a file at an absolute address should only be used for data files, since the monitor will not set up a job for it.



### 3.4.2 Queue tracing

**Command:** `queue [#chan[,]] [<address>[,<lines>]]`

**Default channel:** 0
**Default address:** No default address except to continue tracing the remainder of a queue.
**Default lines:** the previous ‹lines› value, or the height of the window if no previous ‹lines› value. (8 lines if output not to window).

**Action:** traces round a "queue" of addresses, outputing 8 bytes of memory for each address in the queue. The listing stops if the queue wraps round on itself (i.e. ‹address› is met again), if 0 is encountered, or if the number of lines specified have been output (the rest of the queue can be traced by re-issuing the queue command).

### 3.4.3 Leaving the monitor

**Command:** `quit`

**Action:** kills the monitor job, plus any clones started with the **clone** command (see section 3.8). Any jobs loaded with the **load** command will not be removed — use **kill** to remove these if necessary. If you quit the monitor while there are still jobs running which have been started or examined by the monitor, the system may crash if any of the jobs encounter a breakpoint or program error after the monitor job has been deleted.

### 3.4.4 Screen refresh

**Command:** `mode [<mode>]`

**Action:** If no ‹mode› is specified refreshes all current monitor windows. If ‹mode› is 0 sets the screen to default startup screen.

### 3.4.5 Eval

**Command:** `eval [<value>[ [\[<base>]]`

**Default channel:** 0
**Default value:** ?bp
**Default base:** hex

**Action:** evaluates the expression supplied, outputting the result as a 32-bit hex number, if no base is given, or as a signed number in the given base. Note that the default will tell you the address of the base of your current job, and that

```
eval @<address>
```

can be used to evaluate address expressions.

Examples:

```
eval ^12345\2   (convert octal 12345 to binary)
eval 0ef8\8     (convert $0ef8 to octal)
```

Sections 2.3.1 and 2.3.2 show more examples of this command.

### 3.4.6 Saving code

**Command:** `save filename <addr range> [<datasize>]`

**Action:** saves the given range of memory to the given file. If a data size is specified the given file is made an EXECable job, with the given value as its data size. Note that the data size given will be subtracted from the top value in the address range, since this is assumed to be the current data space of the job.

Example:

```
save mdv2_myjob_exe = 100
```

would save from the current job's base pointer to $100 bytes from the top of the job.

### 3.4.7 Setting flags

**Command:** `flags {[no]<flag>}*`

**Action:** sets the default values for flags on all subsequent commands. The default can always be overridden by specifying the flags on the command being used.

Examples:

```
flags nohex norel rn   (output from dump or dis will suppress hex
                        parts. All addresses will be absolute. Regs
                        command will display user registers)
```



### 3.4.8 Allocating heap

The heap allocation commands allow you to allocate space on the common heap, saving the base address of the allocated memory, and to deallocate the space. You can also display all the heap allocated to the current job.

**Command:** `heap <address> <value>`

**Action:** allocates <value> bytes of common heap and stores the base address at the given address.

**Command:** `noheap <address>`

**Action:** returns to the common heap the area of memory whose base address is stored at the given address. Note that this area must have been allocated from the heap.

**Command:** `heap`

**Action:** displays the base address and length of all areas of memory allocated from the heap to the current job. Be wary of using the clone command with this command, since the heap may be changing while the clone is looking at it. This may cause an address trap error in the clone.

Examples:

This example shows how the screen could be saved on an area of heap and then restored from the heap.

```
heap r0 8000            (allocate 32k and store base in reg r0)
move 20000\8000 (r0)    (move 32k from address $20000 (the screen)
                         and store at base address)
.
.
move (r0)\8000 20000    (restore screen)
noheap r0               (deallocate heap)
```

### 3.4.9 Managing channel ids

**Command:** `Channels`

**Action:** displays the channel ids of all channels open for the current job and the address of the channel block on the heap.

**Command:** `nochannel <chan-id>`

**Action:** closes the channel stated. The channel-id is a 32-bit number similar in format to job-id.

## 3.5 PROGRAM DEBUGGING

### 3.5.1 Tracing code

**Command:** `Trace [#chan[,]] [<address>][,<lines>] [[;]flags]`

**Default address:** current program counter (PC).
**Default lines:** 1
**Flags:** [no]pc [no]hex [no]job [no]rel [no]an [no]dn [no]rn [no]dis [no]regs [no] watch [no]break [no]jsr [no]trap

**Action:** The first seven flags are present as trace normally automatically issues implicit **dis** and **regs** commands (unless told otherwise by adding nodis and noregs). The instruction at the address given is disassembled, along with all other instructions just prior to their execution. At the end of the batch of instructions being executed (normally only one for single-stepping), a register dump is produced and you are then able to examine/alter registers as usual. The down-arrow key is set to continue tracing (the disassembly of the first instruction is suppressed to join the display sensibly). The up-arrow key can be used to skip over a single instruction should you not wish to execute it. When tracing the monitor will report program errors (see section 3.5.6), break points (see section 3.5.3) and watch points (see section 3.5.4). For details of the jsr and trap flags see sections 3.5.5.

Examples:

| | |
|---|---|
| `trace` | (single step current instruction, with disassembly before and after, plus register dump) |
| `trace 400,6 nohex` | (trace through 6 instructions starting at $400(bp) with register dump at end of trace. Don't output hex code with disassembled instructions. Number of instructions to execute is 6 from now on) |

### 3.5.2 Unconditional execution

**Command:** `go [<address>]`

**Default address:** current job's program counter (PC).
**Flags:** [no]break



**Action:** releases the current job, setting the priority to $20 if the job has no priority, i.e. is inactive. The job will continue to execute until it is suspended, or a breakpoint (see section 3.5.3.) or program error occurs (see section 3.5.6).

**Break and Watch points**

To simplify the process of debugging programs you are allowed to set "breaks' in the program (to save single-stepping through large amounts of code). You can also make the trace mechanism watch for certain events (e.g. a register reaching a certain value).

### 3.5.3 Breakpoints

**Command:** `break <address>`

**Action:** Set a break point at the specified address. When using the **trace** or **go** command execution will stop at this address. Note that the breakpoint is not actually set in the code until a **go** or **trace** command is issued.

**Command:** `nobreak <address>`

**Action:** clear the break point at the specified address.

**Command:** `breaks [#chan]`

**Default channel:** 0
**Action:** display all current break points.

**Command:** `nobreaks`

**Action:** clear all breakpoints

Examples:

with the code (nohex)

```
0000(bp)      moveq #10,d0
0002(bp)      moveq #7,d2
0004(bp)      add.l d0,d3
0006(bp)      dbra d0,0004(bp)
```

you could issue the following commands:

| | |
|---|---|
| `break 4` | set a breakpoint at 4(bp) |
| `breaks` | list breakpoints (0004(bp) will be listed) |
| `trace 0,10` | start execution of 10 instructions the trace will stop at 0004(bp) with a message of the form |

```
Breakpoint
JB=<job-id> PC=0004(bp) SR=000

trace           continue trace from break
                another break will occur when the program loops
nobreak 4       cancel the breakpoint
trace           10 instructions will be traced without interruption
```

**Note:** Breakpoints cannot be set in ROM, but they can be simulated using watch (see section 3.5.4).

The down-arrow command set for tracing sets all breakpoints, including the one at the current instruction (if any). To continue from a breakpoint when single-stepping the trace command must be issued; down-arrow will not work. (trace sets all breakpoints except at the current pc).

### 3.5.4 Watchpoints

**Command:** `watch[_<size>] <addr range> <value>`

**Default size:** l (long word)
**Action:** watches for a location at the specified address or in the range of addresses being set to the given value. Note that longwords are watched for at four byte intervals, not on all even addresses. Watchpoints can only be used with the **trace** command; the **go** command does not watch them.

**Commands:** `nowatch[_<size>] <addr range>`

**Action:** removes watches at the given address or range of addresses.

**Command:** `watches`

**Action:** lists the addresses of all current watch points.

**Command:** `nowatches`

**Action:** removes all watch points

Examples:

```
watch_w d0 -1
```

would watch for the value of register d0 becoming S????FFFF and would be suitable as a watchpoint on the program above to stop tracing when the loop has terminated (dbra decrements the bottom word of d0 until it reaches −1). To cancel the watchpoint the command



```
nowatch_w d0
```

would have to be issued (the size extension to the basic command must match for a register watchpoint). A breakpoint in ROM can be implemented using

```
watch pc <address>
```

Watchpoints terminate execution prior to executing an instruction (i.e. just after the disassembly). The note on the use of the down-arrow with breakpoints (section 3.5.3) also applies to watchpoints.

### 3.5.5 Skip Tracing

It is useful to skip certain instructions, especially trap calls, whose affects are well known and assumed to be correct. Normally the **trace** command will skip the inner code of a trap call, producing no disassembly for the code and effectively treating the trap as a single instruction. Full tracing then continues again one instruction past the call. If you do wish to trace through the QDOS traps then the trap flag should be turned on when the trace command is being used.

Example:

```
trace (pc),5 trap
```

It is also possible to have the Monitor set a temporary breakpoint just past a jsr or bsr instruction to allow the inner code of these instructions to be skipped as well. However, care should be taken when skipping any of the QDOS subroutine vectors, which can return to addresses other than immediately following the jsr instruction (for example vector S122). In this case you can set your own breakpoints or watchpoints at all possible return addresses. If you find that you have lost control of the monitor (because it is waiting for the temporary breakpoint immediately following the jsr) you can press the CAPSLOCK key 4 times to regain control. This can also be tried at other times when the monitor appears to have "locked".

Example:

With the following code

```
0026(bp) jsr 0084(bp)
002A(bp) moveq #6,d0
002C(bp) subq.l #2,d0
                :
                :
                :
```

the trace command

```
trace 26 nojsr
```

will only produce output of the form

```
0026(bp) jsr 0084(bp)
002A(bp) moveq #6,d0
```

(assuming the jsr completes correctly)
rather than

```
0026(bp) jsr 0084(bp)
0084(bp) ....
```

Note that these flags do not apply when tracing in ROM, so if you decide to trace a QDOS vector using the jsr instruction, then all traps used by this subroutine will be traced.

If you start tracing a subroutine and wish to skip the rest you can easily set a breakpoint, assuming you are not in ROM, by issuing the commands

```
break ((a7))    (set break point at routine return address)
go              (continue until break point)
```

### 3.5.6 Program Errors

The 68008 is capable of generating certain sorts of "exception" (error), some of which QDOS allows the user to trap. The exceptions can be annoying when developing machine-code on the QL as normally the program tends to crash, slowing down the QL considerably, or stopping the machine altogether. The monitor will report the exception which has occurred in order to allow you to correct it and continue execution. The monitor can detect errors on any jobs that have been loaded and started using the monitor or have been EXECed since the monitor was started. Thus, you may get errors on jobs other than the current one. The following are messages which will be issued, with an explanation of the reasons why the exception may have occurred, and possible corrective action. The format of the message output on a program error is shown below, using an illegal instruction error as the example error message.

**Message:** `Illegal instruction`
`JB=<job-id> PC=<address> SR=<status>`

`JB=<job-id>` => the job-id of the job in which the error occurred. This can be any QDOS job which has been started or examined using the monitor and need not be the current job.



`PC =<address>` => value of program counter when exception occurred.

Note that in all exceptions and the register dump the ‹address› is quoted relative to bp if its value is between the contents of bp and tp, unless the flag rel is permanently set off.

`SR=<status>` => value of status register when the exception occurred, in hex plus flag values.

Note that the second nibble (4 bits) indicates the setting of the interrupt mask (and should normally be 0) and that the first nibble indicates the setting of the trace and supervisor-mode bits (8=trace,2=super,A=both).

If an error occurs in a job that is not being traced, the job is suspended and the job header is set to reflect the register values of the job when the error occurred. You could then use the job command to make this job the "current" monitor job, since the error gives the job-id, so that it can be exmained.

**Message:** `Address Trap Error`
`FC=xxx XY @<address> IR=yyyy`

where xxx= 001=> user data memory access
010 => user program memory access
101 => supervisor data memory access
110 => supervisor program memory access
111 => interrupt acknowledge

X= I => instruction in progress
E => exception processing

Y= R => read cycle aborted
W => write cycle aborted

@‹address› => address generated

yyyy= instruction in Instruction Register

The program counter may not be at the start of an instruction (this is why the value in the instruction register is also quoted).

If tracing is in progress then the offending instruction should be in view, otherwise to find the instruction it is suggested that you issue the command

```
dis -10(pc)
```

**Reason:** word and longword data addresses must be at a word aligned (even) address. Hence

```
move.l #1,a0
move.l (a0)+,d0
move   label, a3    (label not word aligned)
```

will cause this exception.

**Corrective action:** alter the contents of the offending address register, or rewrite the program! If you alter the register to its correct value using the poke or . command, you can set the program counter to the instruction that cause the error and continue execution.

**Message:** `Privilege Violation`

**Reason:** certain instructions can only be executed in supervisor-mode, e.g. andi.w #$DFFF,SR or rte.

**Corrective action:** set the supervisor-mode bit in the status register, or add the instruction trap#0 to your program (the QDOS call to enter supervisor-mode).

**Message:** `Illegal Instruction`

**Reason:** an unrecognised opcode has been met (its disassembly is likely to be suspect). (An rts may cause this or other errors if the stack has been mismanaged.)

**Message:** `Division by Zero`

**Reason:** something like divs d1,d0 has occurred when d1 contains zero.

**Corrective action:** ensure that the first argument for divs or divu is non-zero (test first if necessary).

**Messages:** `CHK Instruction Trap`
`Unrecognised Trap`
`Unimplemented Opcode`

**Reasons:** other events which occur in instruction-specific circumstances. Only the sophisticated user is likely to meet these! (no doubt they can cope). It is possible that these will occur as an alternative to Illegal Instruction, if something has gone wrong, causing execution to continue at the wrong address, for example when the correct return address for an rts instruction is not on top of the stack.



NB Unimplemented opcodes may not always be spotted since QDOS has some code on top of the relevant vectors. Any of these events in the region of PC=61?????? may in fact be an unimplemented opcode.

**Message:** `Unexpected Trace Trap`

**Reason:** the trace bit in the status register has been set, but not by the trace command.

## 3.6 CHANNEL MANAGEMENT

The channel management commands are similar to superBASIC's but not identical. By default these commands always refer to the last channel referenced, e.g.

```
regs
cls
```

will clear the register window (#2). (Specific channels can be used of course as with all commands.) To change the default channel for a command use the command name followed by the required channel and a full-stop.

Example:

```
dump #3.    (dump output defaults to channel 3)
```

**Command:** `cls [#chan]`

**Default channel:** last channel used

**Action:** clear last screen used, or given channel.

**Commands:**
```
ink [#chan,] [<colour>]
paper [#chan,] [<colour>]
strip [#chan,] [<colour>]
border [#chan,] [<colour>]
```

**Default channel:** last channel used.

**Defaults:**
ink 7
paper 0
strip 2
border 2

**Actions:** set the colours of various parts of a window. Note that paper sets the strip colour too. Only border widths of 1 are supported.

**Commands:** `open #chan,<device>`
`close #chan`

**Actions:** open/close a QDOS device. Up to 16 channels can be used, numbered 0 to 15. Channel 1 is always opened for input and is the channel from which commands are taken. Thus channel 1 could be opened as a microdrive file to take a set of initial commands when the monitor is EXECed. If a file read error occurs, the file is closed and a standard console device is opened, and no error is reported. Bad command lines will be ignored by the monitor, so files of commands should be carefully constructed.

**Command:** `window [#chan]`

**Default channel:** last channel used

**Action:** this command allows you to alter the position and size of the last window accessed, using the cursor control keys. A flashing cursor appears in the channel 1 window when the command is entered. Use the up-arrow, down-arrow, left-arrow and right-arrow keys to move the relevant window. Use the CTRL key with these keys to move the window by 10 pixels at a time. When you have finished moving/altering the window just press ENTER to return to the monitor. The size of the window will then be displayed.

Examples:

To output disassembled code to a printer you could use

```
open #10, ser1c
dis #10,0,20
```

To open a file from which the monitor takes commands you could use

```
open #1, mdv1_boot_mon
```

This could be useful if you wanted to design your own screen layout. Two command files are supplied with the monitor microdrive cartridge (boot_mon and demo_mon).

Note that the open command always implicitly closes the previous device open on the channel — the device can be closed with the close command if necessary.

To open a new window and change its size

```
open #5, scr
window              (can now alter/move new window to suit)
(use cursor keys here)
(ENTER)             (return to monitor prompt)
```



## 3.7 JOB MANAGEMENT

There is not enough space to explain the concept of a job in full in this manual, so certain parts of this chapter will assume that the reader knows what a job is. Some explanation is supplied in the overview. In addition to what is said there you should realize that jobs are normally programs, and that these programs are executed "concurrently" by QDOS. (i.e. you could start the one-line superBASIC program

```
repeat loop:at 0,0:print date$
```

yet still be able to use the monitor without apparent interruption.)

### 3.7.1 Listing jobs

**Command:** `jobs [#chan]`

**Default channel:** 0
**Flags:** [no]rel [no]job

**Action:** lists all jobs not dependent on the monitor giving the job_id, the owner, the base of job (the job header is $68 bytes back from the base in QDOS version 1.03 and earlier versions), the priority and the job length.

### 3.7.2 Altering the "current" monitor job

**Command:** `job <job-id>`

**Flags:** all those that apply to a register dump and [no]regs

**Action:** Makes the given job the "current" monitor job. Automatically displays the current state of the jobs registers. This allows the job to be traced. All register dumps using the **regs** command will refer to this jobs registers. Note that the **load** command automatically sets the "current" job to the one it loads. Making a job that is currently running the "current" job causes the job to be suspended. BP and TP are also set. Note that because superBASIC moves from time to time BP and TP are continuously updated and will not be alterable if the current job is superBASIC (job-id = 0).

### 3.7.3 Suspending jobs

**Command:** `suspend [<job-id>]`

**Default:** current monitor job

**Action:** Suspend the job with the given ‹job-id›.

### 3.7.4 Releasing jobs

**Command:** `release [<job-id>]`

**Default:** current monitor job

**Action:** release the job with the given ‹job-id›. Note that if the job has priority 0 then releasing it will not cause it to start executing, but will only remove it from its suspended state. In this case use the priority command to increase the job's priority. Note also that if you are releasing the current monitor job you can use the go command, which also sets the priority to $20, if it is found to be 0.

### 3.7.5 Altering the priority of jobs

**Command:** `priority [<value> [<job-id>]]`

**Default value:** $20
**Default job-id:** current job

**Action:** sets the priority of the given job to the given value. If the priority was 0 and the job is not in a suspended state it will start executing. Similarly, if the priority is set to 0 while the job is running, this will stop it.

Example:

```
job 40006       (suspend/examine job and registers)
.
.d0 0           (alter registers)
.
.
go              (start job again)
```

### 3.7.6 Killing jobs

**Command:** `kill <job-id>`

**Action:** abort execution of the job stated (any job waiting for the completion of the job named will receive a "not complete" error). This command should be used, before exiting the monitor, to force remove all jobs loaded using the **load** command, unless the jobs contain code to delete themselves. If you kill the current job then certain commands, including **regs** and **trace** will give an "invalid job" error, until you change to a new job by using the **load** or **job** commands. Note that `k .` can be used to kill the current job.



## 3.8 CLONING MONITOR COMMANDS

**Command:** `clone [<command>] {:<command>}`

**Action:** if no ‹command› is specified this command lists all clones in a form similar to the jobs command. Otherwise it sets up a job to repeatedly execute the given command(s). This command can be extremely useful for monitoring the progess of jobs without explicitly tracing them. For example, if you suspected that a job was looping somewhere, you could clone the **regs** command so that the registers for the job were continually updated on the screen.

Examples:

```
flags noregs     (stop all implicit register dumps)
clone regs       (start up continuous display of registers)
```

sets up a job which continuously issues the **regs** command. This means that window 2 (assuming this is still the register window), is effectively "jammed" for all other purposes, since it is homed and written to by the clone on a never-ending basis. (The bottom of the window is always cleared just before the window is homed to cater for command whose output varies in length over a period of time, e.g. during clone jobs.)

```
clone jobs #5
```

would give an up-to-date picture of all the jobs to channel 5, which you might have opened as a screen which doesn't interfere with the rest of the monitor screens.

```
clone  dump #5,(a1),2
```

would continually display the memory pointed at by the register a1. This would be useful for monitoring the contents of a buffer.

```
clone  eval #6,@0(a6,a1)
```

would continually ouput to channel 5 the address of the top of the superBASIC stack, if the job-id is 0.

To control clones you can use the normal job-control commands, i.e. suspend, release, priority, kill. The clone command supplies a list of job-ids, which are used in the normal way. Clones are sub-jobs of the monitor and are therefore automatically deleted when the quit command is issued.

# CHAPTER FOUR
# SUMMARY

## 4.1 GENERAL COMMAND STRUCTURE

```
command [#<channel>[,]][<parameters>[[;]<flags>]
```

All commands can be abbreviated (an ambiguous abbreviation causes the first in the following list to be take). (Relevant flags are listed in more detail in section 4.3.) If more than one command is specified on a single input line, a colon should be used as a command separator.

| Command: | Default: |
|---|---|
| help | |
| dis [<address>][,<lines>] | dis #<dis_chan>,<br><last-address>, <last-lines><br>(lines=0 => use window height) |
| dump [<address>][,<lines>] | dump #<dump_chan>,<br><last-address>, <last-lines><br>(up- and down-arrow set up after dis/dump) |
| trace [<address>][,<lines>] | trace (pc),1 notrap<br>(down-arrow => next instruction(s))<br>(up-arrow => skip instruction)<br>(stops on lines, breakpoint or watchpoint) |
| regs | |
| jobs | regs #<regs_chan>;norn |
| job <job-id> | |
| go [<address>] | go (pc) |
| load <device> [<address>] | |
| save <device> <range> [<value>] | |



| Command: | Default: |
|---|---|
| find[_<size>] <range> <value> | find_b |
| poke[_<size>] <range> [<value>]* | poke_b <range> 0 |
| .[_<size>] <range> [<value>] | ._l <value> 0 regs |
| queue [<address>][,<lines>] | queue <last-address>,<br><last-lines> |
| quit | |
| move <range> <address> | |
| compare <range> <address> | |
| break[s] [<address>] | |
| nobreak[s] [<address>] | |
| watch[es][_<size>][<range> <value>] | watch_l |
| nowatch[es][_<size>] [<range>] | nowatch_l |
| suspend [<job-id>] | |
| release [<job-id>] | |
| priority [<value> [<job-id>]] | priority 20 <current job> |
| kill <job-id> | |
| window | window #<last-chan> |
| cls | |
| open <device> | |
| close | |
| ink [<colour>] | ink #<last-chan>,7 |
| paper [<colour>] | paper #<last-chan>,0 |
| strip [<colour>] | strip #<last-chan>,2 |
| border [<colour>] | border #<last-chan>,2 |
| mode [<mode>] | |
| clone[s] [<command>] | |
| channels | |
| nochannel <channel-id> | |
| heap [<address> <value>] | |
| noheap <address> | |
| eval [<value>] [\[<base>]] | eval ?bp \$10 |
| flags { [no]<flag> }* | |

### 4.1.1 Parameter Syntax

The constructs used above are defined as follows:

```
<value>     =[$]<hex_number> | &<decimal_number>
            | ^<octal_number> | %<binary_number>
            | "<string>" | '<string>' | ~<value>
            | <value>| -<value> | <value>+<value>
            | <value>-<value> | <value>*<value>
            | <value>/<value> | <value>! <value>
            | <value>&<value> | <value><<value>
            | <value>><value> | <value>^<value>
            | <base>%<value> | ?<address> | @<address>
```

The operators above which are not self-explanatory are:

! (bit-or) & (bit and) < (shift left)
> (shift right) ~ (not) ^ (mod)
% (allows the given value to be specified in the given base)

[ and ] are used to alter precedence, levels are as follows:

```
+,- (monadic),~,&,$,^,%          (value operands)
+,- (dyadic),*,/,!,& (diadic),<,>,^ (diadic),% (diadic)

?,@                                  (value operands)
                                     (address operands)

<address>   =<value> {==<value>(bp) if <value> < $10000}
            |<value>£ {==forced absolute}
            |<register>
            |[<value>](<address>{,<address>[.w|.l]+)
```

(last of these includes (rn) xxxx(rn) xx(rn,rm) etc. Spaces are stripped throughout, except before the "(" in, say 99(a0) which causes ambiguity in some commands.)

```
<register>  =d0..d7|a0..a7|pc|sr|r0..r7|bp|tp

<range>     =<address1>[,<address2>] |
             <address>\<value> | =
            ( = means from job base to top of job)

<size>      =b|w|l

<channel> | <lines> = <decimal_number>
<mode> | <colour> | <job-id> | <channel-id> | <base> =
                                              <value>
```



## 4.2 CHANNELS

The main text of this manual probably gives the impression of a set of actual default channels for each command. It is more correct to say that a logical channel is attached to each command, and that an actual channel is then associated with each logical channel:

| Logical Channel | Actual channel on startup |
|---|---|
| ‹last_chan› | 0 |
| ‹dump_chan› | 0 |
| ‹regs_chan› | 2 |
| ‹dis_chan› | 3 |

Most commands use ‹dump_chan›. To alter the logical to actual relationship, issue any command directed to the logical channel with a trailing full-stop. e.g. dis #0. or help #3. (these would force even implicit references to the relevant windows to alter, i.e. the implicit disassembly within a trace would now be directed to channel 0.)

## 4.3 FLAGS

These can be set globally using the flags command, or locally on the command itself. Flag names cannot be abbreviated (ambiguity with parameters would occur if this was allowed).

| Flag | Commands | Controls |
|---|---|---|
| rel | any command which outputs addresses | output of address in bp-relative form |
| job | ditto | output addresses as xxxx + job not xxxx(bp) |
| pc | dis,dump,regs | output of location/program counter |
| hex | dis,dump | output of code/hex |
| asc | dump | output of ascii |
| an | regs | output of address registers |
| dn | regs | output of data registers |
| rn | regs | output of user registers (normally off) |
| dis | trace | implicit disassembly |
| regs | trace, load, job | implicit register dump |
| trap | trace | skip-tracing of trap calls (normally off) |
| jsr | trace | skip-tracing of jsr/bsr instructions |
| break | trace,go | setting of breakpoints |
| watch | trace | examination of watchpoints |

# INDEX