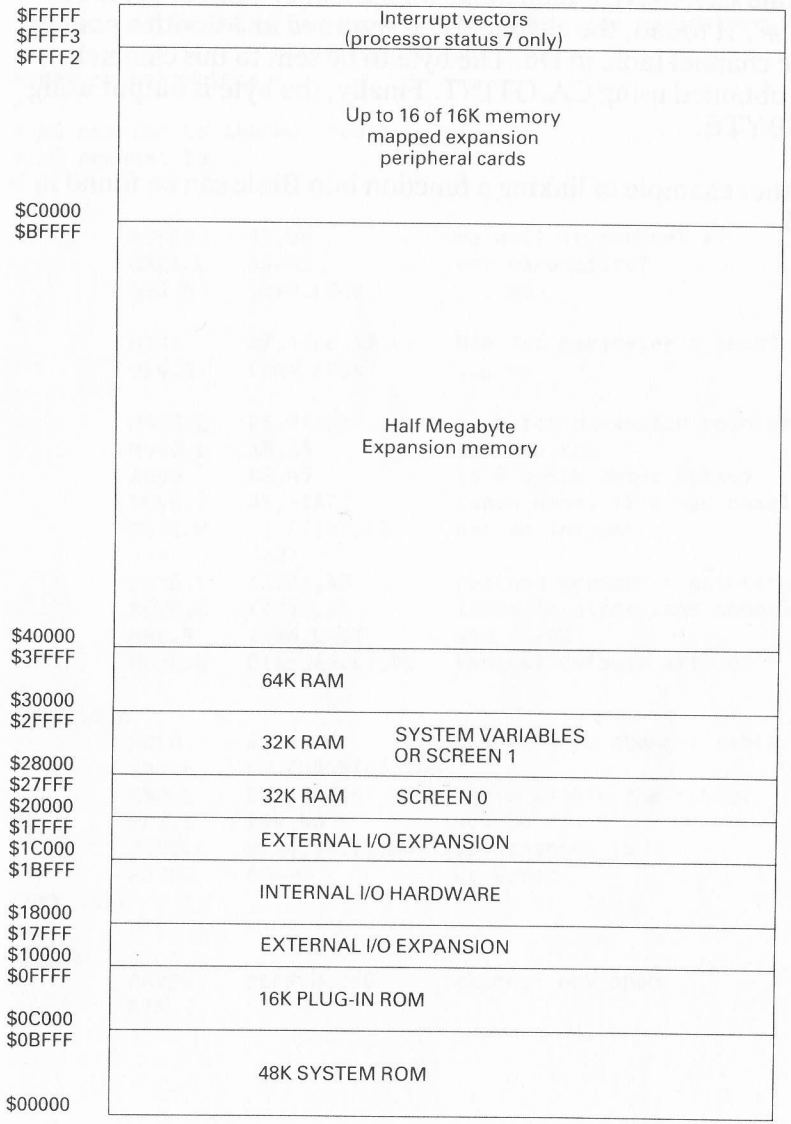
Appendix A – The memory map

This appendix illustrates the memory map of the QL as seen from a hardware point of view. The memory as seen by QDOS is discussed in section 3.2.



Appendix B – 68008 Instruction Set Summary

A more detailed analysis of the 68008 microprocessor and it's instruction set can be found in chapter 2.

Instruction Set

Mnemonic	Description
ADBC	Add Decimal With Extend
ADD	Add
AND	Logical And
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
B_{CC}	Branch Conditionally
BCHG	Bit Test and Change
BCLR	Bit Test and Clear
BRA	Branch Always
BSET	Bit Test and Set
BSR	Branch to Subroutine
BTST	Bit Test
CHK	Check Register Against Bounds
CLR	Clear Operand
CMP	Compare
DB_{CC}	Test Condition, Decrement and Branch
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR	Exclusive Or
EXG	Exchange Registers
EXT	Sign Extend
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link Stack
LSL	Logical Shift Left
LSR	Logical Shift Right

Mnemonic	Description
MOVE	Move
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral Data
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NOP	No Operation
NOT	One's Complement
OR	Logical Or
PEA	Push Effective Address
RESET	Reset External Devices
ROL	Rotate Left without Extend
ROR	Rotate Right without Extend
ROXL	Rotate Left with Extend
ROXR	Rotate Right with Extend
RTE	Return from Exception
RTR	Return and Restore
RTS	Return from Subroutine
SBCD	Subtract Decimal with Extend
S_{CC}	Set Conditional
STOP	Stop
SUB	Subtract
SWAP	Swap Data Register Halves
TAS	Test and Set Operand
TRAP	Trap
TRAPV	Trap on Overflow
TST	Test
UNLK	Unlink

Variations of Instruction Types

Instruction Type	Variation	Description
ADD	ADD	Add
	ADDA	Add Address
	ADDQ	Add Quick
	ADDI	Add Immediate
	ADDX	Add with Extend
AND	AND	Logical And
	ANDI	And Immediate
	ANDI to CCR	And Immediate to Condition Codes
	ANDI to SR	And Immediate to Status Register
CMP	CMP	Compare
	CMPA	Compare Address
	CMPM	Compare Memory
	CMPI	Compare Immediate
EOR	EOR	Exclusive Or
	EORI	Exclusive Or Immediate
	EORI to CCR	Exclusive Or Immediate to Condition Codes
	EORI to SR	Exclusive Or Immediate to Status Register

Instruction Type	Variation	Description
MOVE	MOVE	Move
	MOVEA	Move Address
	MOVEQ	Move Quick
	MOVE from SR	Move from Status Register
	MOVE to SR	Move to Status Register
	MOVE to CCR	Move to Condition Codes
	MOVE USP	Move User Stack Pointer
NEG	NEG	Negate
	NEGX	Negate with Extend
OR	OR	Logical Or
	ORI	Or Immediate
	ORI to CCR	Or Immediate to Condition Codes
	ORI to SR	Or Immediate to Status Register
SUB	SUB	Subtract
	SUBA	Subtract Address
	SUBI	Subtract Immediate
	SUBQ	Subtract Quick
	SUBX	Subtract with Extend

Appendix C – Plug in ROM specification

Introduction

This appendix will be of interest to those who wish to add their own utilities to the QL in ROM format. It will also be of interest to anyone who wishes to find out how QDOS links in user supplied routines in ROM.

The ROM cartridge slot can be found on the back of the QL. Plug in ROMs connect directly onto the main QL printed circuit board via a 30 way double sided card edge connector. This allows ROMs of up to 16K bytes in size to be plugged in. The cartridge could contain, for example:

1. Software which would replace SuperBasic such as new languages like LISP, PASCAL, BCPL, assemblers, utilities for debugging programs, specialised applications software etc.

2. Software which can be linked into SuperBasic to add new procedures.

—	a	1	b	+5v
A12	a	2	b	A14
A7	a	3	b	A13
A6	a	4	b	A8
A5	a	5	b	A9
SLOT		6		SLOT
A4	a	7	b	A11
A3	a	8	b	ROMOEH
A2	a	9	b	A10
A1	a	10	b	A15
A0	a	11	b	D7
D0	a	12	b	D6
D1	a	13	b	D5
D2	a	14	b	D4
GND	a	15	b	D3

Figure C.1 – ROM slot signals

Hardware connections

A variety of useful signals are provided from the ROM slot. The 16 address lines allow up to 64K bytes of ROM to be addressed, but in practice there is only a 16K byte slot available in the memory map (see Appendix A). The data lines allow bytes to be transferred from the ROM to the 68008 microprocessor. ROMOE is an active low signal which enables the ROM for output whenever the 68008 tries to read from it. Finally, there are the two power supply lines at 0 and +5 volts respectively. The layout of the signals is illustrated in figure C.1 below.

ROM software specification

There are several aspects of plug in ROM software. Firstly, there must be a special recognition sequence which QDOS can test. If the sequence is there, it is fairly reasonable to assume that the ROM is indeed plugged into the back of the computer. Secondly, having ascertained that there is a ROM there, QDOS must be able to find all of the routines which it wants to use, and link them into the system.

The special recognition code which should be placed at the start of the ROM is 4AFB0001 Hex followed by the driver header. The header points to a list of BASIC procedures and functions within the ROM, together with the address of the initialisation routine. There is also a string to identify the ROM. The header is formatted as follows:

00 long word recognition flag (4AFB001 Hex)
04 pointer to list of BASIC functions & procedures
06 pointer to initialisation routine
08 string identifying the ROM

All pointers are relative to the base address of the ROM. The string identifying the ROM should be in the form of a character count (word) followed by the ASCII characters of the device description(s). This string should be terminated with a <LF>. It is recommended that the number of characters should be limited to 36.

The list of BASIC procedures and functions is in the form:

word number of procedures

then for each procedure:

word	pointer to routine within ROM
byte	length of name of procedure
characters	making up name

followed by:

word	equal to zero at end of procedures
word	number of functions

then for each function:

word	pointer to routine within ROM
byte	length of name of function
characters	making up name

finally followed by:

word	equal to zero at end of functions

code for routines can then start here and fill up the rest of the ROM.

The sequence of events which occur when the machine is booted is as follows. First of all, the ROM recognition flag is checked. If a ROM is present, the machine will link in the additional BASIC procedures from the ROM. The number of procedures and functions is used to reserve internal table space in RAM. If the average length of the procedure or function names exceeds 7 then the effective number of procedures and functions is calculated as (total number of characters + number of functions or procedures +7)/8. After linking into BASIC, the initialisation routine is called (in user mode). This routine must not modify A6 at all, and must restore A0 (the initial window ID) and A3 (the pointer to the ROM) before returning. Up to 128 bytes can be used on the USER stack.

Finally, for more information about writing device drivers, reference should be made to chapter 9.

Appendix D – Microdrive format

This appendix is essentially a collection of useful facts about the QL microdrives. It gives details about the way in which files are spread over the continuous tape loop and why. Reading this appendix will provide a much deeper understanding of microdrive operation as the machine sees it.

D.1 Microdrive format

The microdrive medium is a long loop of high quality magnetic tape. When the drive motor is running, any particular point on the tape will pass the read/write head once every 7 seconds. Some way of measuring where on the tape loop a particular piece of data can be found is required. Ideally, data would be written over the entire length of the tape. In practice, this is not implemented for several reasons. First of all, there is almost certain to be some minor blemish on the tape where it will not be possible to record data reliably. Secondly, if one wanted to modify a byte, the erase head would have to be turned on instantly before the byte went passed, then turned off again at the end of the byte to ensure that no other bytes were erased.

To produce a practical and relatively reliable system, the loop is split up into sectors. Each of these sectors contains 512 bytes of data. When a microdrive cartridge is first formatted, small markers (sector headers) are recorded around the loop. The space in between the sector headers is filled with dummy data to check that data can be recorded properly on that part of the loop. At the beginning and end of each sector, there is a certain amount of spare space to ensure that the sector can be modified without affecting adjacent sectors, even if it is used in another drive where the motor speed is different. To read or modify a single byte, the whole of a sector must be read in. Once the appropriate byte has been processed, and perhaps modified, it is then necessary to write the entire 512 byte sector back onto the microdrive.

Figure D.1

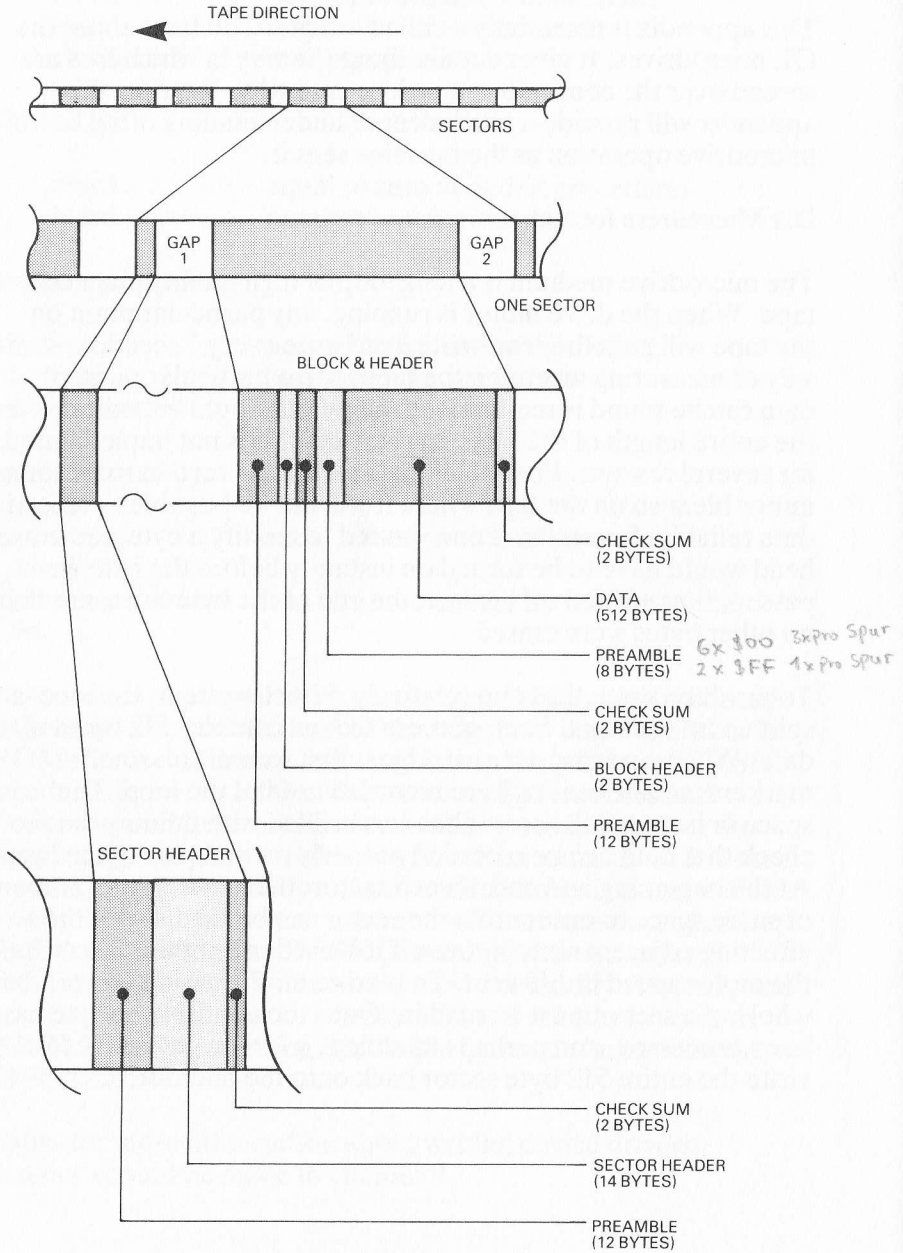


Figure D.1 – Microdrive sector format

There are four recognisable elements recorded on a microdrive:

1. Preamble/Sync

The preamble/sync consists of at least 5 bytes of zeroes. The first two bytes allow the preamble to be recognised. Once recognised, the following three bytes are used to lock the PLL (phase locked loop). The preamble is followed by one byte of ones ($FF) on each track (there are two tracks) to synchronise the read shift registers to a byte boundary.

The standard preamble at the beginning of a sector is 10+2 bytes long. A special 6+2 byte preamble is used within a block. This idles the PLL so that the header of a block can be read separately from the main part of the block.

2. Data

The whole purpose of the microdrive is to store useful data, so this is more important than the other elements (but cannot reliably exist by itself). Data is recorded in multiples of 2 bytes. There are three types of data in a standard sector format. They are the sector header ($FF, sector number and 10 byte medium name), the block header (file number and block number), and block (512 bytes) of a file.

3. Checksum

Each block of data has a special word after it. This word is the checksum, and allows the data to be verified. If the checksum for the read data is not equal to the stored checksum, this normally means that the data has been misread. The checksum is quick to calculate, is proof against missing bits (rotation of apparent data bytes) and is effective against bit patterns of all zeros and all ones. The checksum is calculated as follows:

1. preset the checksum to $0F0F

2. for each byte, add the byte to the checksum

3. record or check the checksum, low byte first then high byte

4. Gap

So that data can be partially overwritten (ie. the whole microdrive cassette doesn't have to be re-recorded every time one byte is changed), gaps are used. There is a gap to separate the sector header (which is only written when the medium is formatted) from the data in the sector (which is rewritten as required). There is also a gap between the end of one sector and the start of the next.

The minimum length required for the gap is calculated as follows. The worst case erase lead time (2660 us) plus the time required to turn erase off (20 us), plus the uncertainty in timing of the write and format procedures (always less than 160 us), plus (for the gap between sectors only – the sector header is never rewritten) the maximum variation on the length of the block written, including the pre-block gap (10% of the total length).

With this length of gap, it is possible for parts of the format block to remain after a sector has been rewritten. This problem is combatted by carefully controlling the content of the format block. This then ensures that no part of the tail end of the format block can be mistaken for a sector header.

The sector header

The sector header is 14 bytes long. The first byte is the sector header flag and is set to $FF. The second byte contains the sector number. The sectors are numbered in descending order down from just below 255 (this depends on the actual length of tape in the cartridge) to 0.

The following ten bytes contain the medium name which can therefore be up to ten characters long. If the name is less than ten characters long, the bytes are padded with spaces. The thirteenth and fourteenth bytes contain a 16 bit random number.

The block header

The block header consists of two bytes. The first byte is a file number or flag. The file number is in the range $00 to $F0 for ordinary files. There are two special file numbers. $F8 is reserved for the microdrive map and the flag vaue $FD indicates that this is

a vacant or bad block. The second byte contains the block number ($00 to $FE). Block numbers start from zero, so bytes 0 to 511 of a file are in block 0, bytes 512 to 1023 are in block 1 and so on.

The block

Blocks are 512 bytes long and contain the data for a file. The data does not start immediately after the end of the block header, but has a short preamble to allow the higher level software to do a bit of processing after reading the block header and before reading the block.

Special blocks

Sector 0 contains the only special block on the medium. This block contains the sector map, and is designated as file number $F8 block 0.

The block contains 255 pairs of bytes. Each pair contains the file number of the file occupying that sector plus the block number within that file. A file number of $FD indicates that the sector is vacant whilst a file number of $FF indicates that the sector is faulty and should not be used. The last byte of the sector map block contains the number of the most recently allocated sector on the medium.

File structure

There is a 64 byte header at the start of every file. This is normally of no concern to the user since the higher level software hides it from ordinary applications.

The format is:

$00	file length
$04	file access key
$05	file type
$06	8 bytes of file type dependent information
$0E	filename as 2 byte character count + ASCII characters
$34	update date – not implemented in QDOS V1.03
$38	reference date – not implemented in QDOS V1.03
$3C	backup date – not implemented in QDOS V1.03

The directory

This is file 0, and is rather a special file because it holds copies of the file headers of all the other files on the medium. The header of file 0 (itself) starts at byte 0 of the file, the copy of the header to file 1 starts at byte 64 and so on. To delete a file, the file length and the name length in the copy of it's header are deleted. This means that the file could be recovered by a recover utility which replaced these two pieces of information.

Sector allocation

The microdrives have a finite start and stop time. It is therefore necessary for the motor to get up to speed before any data is written or read. There are two interleaving factors which should be accomodated as best as possible (this depends on how full the microdrive is). First of all, the first block of a file should be positioned so that there are 20 sectors between the most recently allocated sector and itself (this is to allow for the spin operation which checks that the medium hasn't changed). The second factor is for a file which is more than one block long. Twelve sectors are then skipped between successive blocks of the file. This allows the motor to stop and restart between blocks.

The sector structure on a microdrive

Description	no. of bytes	time	total time
Preamble	12 bytes	480 us	
Sector header	14 bytes	560 us	1040 us
Checksum	2 bytes	80 us	1120 us
Gap 1		3600 us	4720 us
Preamble	12 bytes	480 us	
Block header	2 bytes	80 us	560 us
Checksum	2 bytes	80 us	640 us
Preamble	8 bytes	320 us	960 us
Record	512 bytes	20480 us	21440 us
Checksum	2 bytes	80 us	21520 us
Gap 2		5520 us	27040 us
	Total	**31760 us**	

These times give 225 + 5% sectors per medium.

Special sector structure

The format utility uses a special sector structure to ensure that the entire region which could potentially be written to is checked. This special structure requires that the verified region of the tape should overlap the whole of the potential write region.

The sector header is left in its standard form because it isn't rewritten. However, Gap 1 is shortened so that the written block overlaps the region which would normally be occupied by the preamble. The following block is extended by adding some 86 bytes to the end. The block is filled with $AA55 (binary 1010 1010 0101 0101) except for the word which is 512 bytes from the start, ie. the standard checksum. The test structure is:

Description	no. of bytes	time	total
Preamble	12 bytes	480 us	
Sector header	14 bytes	560 us	1040 us
Checksum	2 bytes	80 us	1120 us
Gap 1	>2840 us	2840 us	3960 us
	(3600−10%−400 (preamble length))		
Preamble	12 bytes	480 us	
Format bytes (EE)	610 bytes	24400 us	24880 us
Checksum	2 bytes	80 us	24960 us
Gap 2	>2840 us	2840 us	27800 us
	Total	**31760 us**	

The extension is designed to take about 10% of the nominal time from the start of gap 1 to the end of a standard sector (2620 us). To this must be added the time by which Gap 1 has been shortened (760 us) plus the time uncertainty (160 us) which totals 3440 us, the equivalent of 86 bytes.

Error types

The following six errors are those which are most likely to occur when using microdrives:

1. Soft read errors

This type of read error can be caused when noise or interference causes false transitions during the reading of a block. One example of noise is that of a piece of dust temporarily distorting the signal as the tape traverses the read head. When the tape comes round again, the dust will (hopefully) have gone and the data can be read correctly. Soft read errors can also be caused where the data has been badly recorded and the signal for a '1' may be misinterpreted as that for a '0'. Soft read errors can always be recovered eventually – if recovery is impossible, it must be a hard error.

2. Soft write errors

Soft write errors are said to have occured if if a block which was recorded cannot be read, but re-writing the block makes it readable.

3. Undetected errors

It is possible for a read error to occur, but for the checksum still to be correct. The error condition cannot then be detected and erroneous data will be read into memory.

4. Hard errors

Hard errors will occur where there are defects in the medium (the splice, dirt or an uneven magnetic coating). The formating procedure is designed to prevent suspect parts of the tape being used for data storage.

5. Interdrive

If a medium is formatted or written on one drive which is marginal, it may not be possible to read it on another drive.

6. Degradation

After a medium has been used for some time, the number of hard and soft errors which it produces will start to increase. This is analogous to the degradation in sound reproduction quality from a sound cassette tape which has been played over and over again. At this stage, the remaining data should be copied onto a new cassette and the old one should be scrapped.

Error precautions

The following precautions are taken to try to ensure that data is correctly stored on microdrives:

1. Bad sector map

A map of good sectors is created by the formatting procedure. When the medium is formatted, a dummy block is written into each sector. The sectors are then verified. Several precautions are taken to make this process as reliable as possible.

a) When the test blocks are written, a compound block is written which is longer than the length of a standard block. These blocks ensure that if a block is re-written on a drive with a different motor speed then the whole of the length occupied by this new block has already been checked.

b) Every sector is checked twice. If an error occurs on either check, the sector on which it occured will be mapped out.

c) The format process writes 255 sectors in decreasing order. The low number sectors overwrite the high number ones. Finally, when sector 0 is written there is the highest block which hasn't been overwritten next to it. If these two sectors are too close, it could cause problems, so the highest sector available is also mapped out to ensure good reliability.

2. Block verify

Every block that is written is verified when it next comes round (7 seconds after it was written). This means that the written block's data must remain in memory until it has been verified. If the verification fails, the block is re-written at the same location.

3. Checksum

There is a sixteen bit checksum for each of the headers and data blocks. There is therefore a one in 64K chance (if the checksum is purely random) that a read error will pass by undetected.

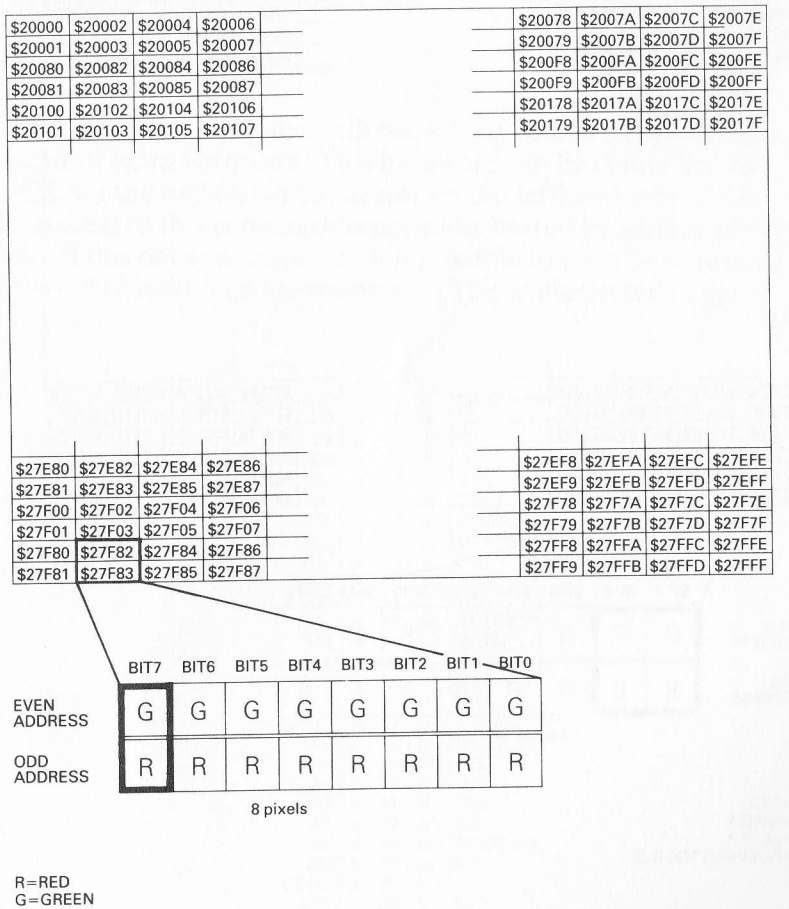
4. Retry

The operating system will only abandon a transaction after 8 passes (detected on sector 0) of the medium occur without any successful operation being performed. A read operation will be retried 8 times before giving up. A write/verify operation will be retried 4 times before giving up.

Appendix E – Screen layout

MODE 4 Screen layout

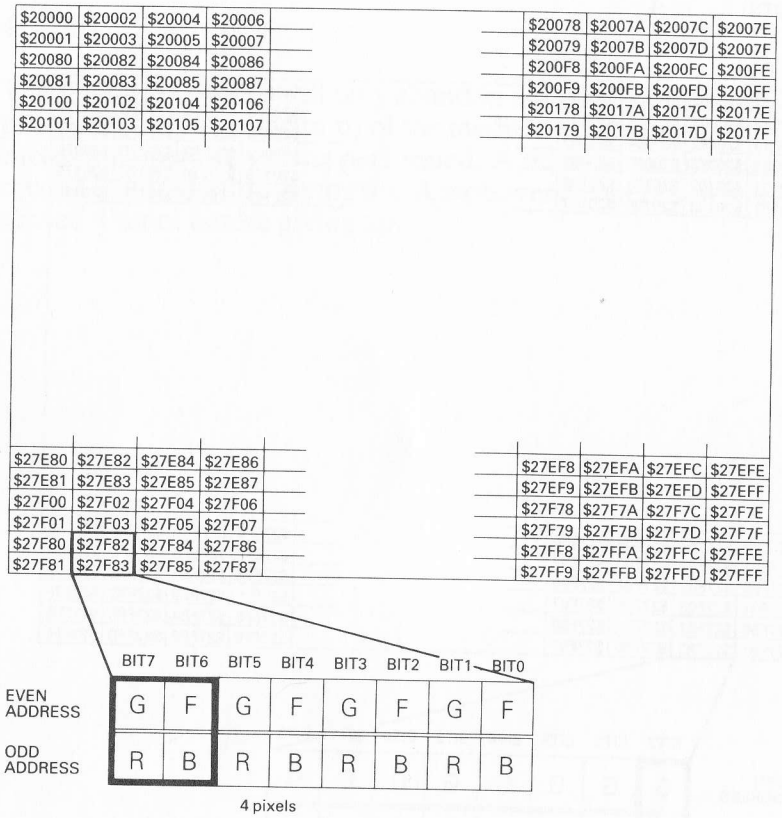
Graphics 512x256
Colours 4



R=RED
G=GREEN

MODE 8 Screen layout

Graphics 256x256
Colours 8 + flashing



R=RED
G=GREEN
B=BLUE
F=FLASHING TOGGLE

Appendix F – Hardware expansion specification

This appendix covers the basic philosophy behind QL hardware expansion. It is intended that this information should be supplemented by a good book on interfacing to the 68008 microprocessor. Note that details of the ROM expansion are included in appendix C and not here.

General aspects of expansion

The QL has been designed with the facility to add a considerable amount of extra hardware. This hardware can be connected to the QL via the expansion connector on the lefthand side of the QL. Access to this connector is normally barred by a black plastic cover. If this cover is removed, it is possible to see a 64-way male DIN–41612 indirect edge connector. This is illustrated in figure F1.

Figure F1 – Connections to the expansion connector

Note that all signals followed by an **L** are active low.

For a description of these signals and their operation, reference should be made to a suitable 68008 databook.

All of the 68008 signals come out to the expansion connector, so it is possible to produce virtually any type of peripheral for the QL. These peripherals will be located on plug-in cards. It is only possible to add one such card to the standard QL. In order to allow for greater flexibility, it is intended that a QL expansion module will become available at some later date. This expansion module will provide it's own power supply and a motherboard with several slots on it to accept a variety of expansion cards as these become available.

Signal description

Output signals

A0 to A19	68008 address lines
ASL	address strobe (indicates valid address)
RDWL	Read/ write
DSL	Data strobe
BGL	Bus grant
DSMCL	*Contended* data strobe (master chip)
CLCKCPU	CPU clock
E	Enable signal to all 6800 type peripherals
RED	Video output red
BLUE	Video output blue
GREEN	Video output green
CSYNCL	Video composite sync signal
VSYNCH	Video vertical sync
ROMOEH	ROM output enable
FC0	Processor status
FC1	Processor status
FC2	Processor status
RESETCPUL	Reset CPU

Bidirectional signals

D0 to D7	Data bus lines

Input signals

DTACKL	Bus arbitration
BRL	Bus request
VPAL	Valid peripheral address
IPL0L	Interrupt priority level 0
IPL1L	Interrupt priority level 1
BERRL	Bus error
EXTINTL	External interrupt
DBGL	Data bus grab

Miscellaneous

SP0 to SP3	Peripheral select lines (up to 16 peripherals)
VIN	9 volts DC nominal (7 volt min.) at 500mA max.
VM12	−12 volts supply
VP12	+12 volts supply
GND	0 volts supply (signal and power supply ground)

Types of expansion cards

Peripheral cards fall into one of two general categories. They are either memory add-ons or general peripheral cards.

The memory map between $40000 and $BFFFF has been allocated purely for the addition of memory (so up to half a megabyte is allowed!). The memory must be contiguous from $40000 upwards. Only one of these add-on memory cards should ever be plugged in at any one time. QDOS will look to see if this memory is available when the machine is powered up.

The peripheral cards are quite interesting. Each can contain a ROM which holds the device drivers (see section 9) for the various bits of hardware on the card. All device drivers have to be position independent because of the way in which the boards are allocated a position in memory.

Allocating memory to expansion cards

The QL edge connector has four signal lines SP0, SP1, SP2 and SP3. These are all grounded on the main board. Each expansion board should contain a piece of circuitry to add 1 to this number, then output the new number to the next card in the chain. Each card occupies the memory slot of 16 kbytes at $C0000 + $4000*n where n is the number input to that particular card. Now it becomes apparent why all device drivers must be position independent. When several boards are plugged in, the expansion cards could be located anywhere in the upper part of memory.

Interfacing to the expansion bus

The 68008 does not have complete control of the bus at all times. So that the information to the video display can be regularly transferred from the memory to the video circuitry, the video ULA holds the bus by using the DTACKL signal. Memory access is controlled by DSL only, and ASL is not used.

The memory map can be considered in terms of the logical states of the upper address lines. To help with visualising the decoding necessary, refer to the table below:

A19	A18	A17	A16	A15	A14	Address range	
0	0	0	0	0	X	00000–07FFF	System ROM (first 32K)
0	0	0	0	1	0	08000–0BFFF	System ROM (extension 16K)
0	0	0	0	1	1	0C000–0FFFF	Plug-in ROM up to 16K
0	0	0	1	0	X	10000–17FFF	Reserved
0	0	0	1	1	0	18000–1BFFF	On board registers and I/O
0	0	0	1	1	1	1C000–1FFFF	Reserved
0	0	1	X	X	X	20000–3FFFF	Standard 128K RAM
0	1	X	X	X	X	40000–7FFFF	Quarter megabyte expansion RAM
1	0	X	X	X	X	80000-BFFFF	Quarter megabyte expansion RAM
1	1	X	X	X	X	C0000-FFFFF	Hardware expansion

On an unexpanded QL, address lines A18 and A19 are never used. When peripheral cards are added, they must disable the circuitry on the main QL board whenever they recognise a valid address to themselves. This can be effected by pulling the signal DSMCL *high*. It is recommended that a fast switching NPN transistor be used to do this.

All signals on the bus can be loaded by up to two low power TTL loads only. DTACKL (data transfer acknowledge) or VPAL (valid peripheral address) must be supplied by peripheral cards as required. DSMCL (master chip data strobe) must be disabled by peripheral cards.

The expansion module provides extra line buffering. The data bus buffers within this module are enabled whenever A18 or A19 is high, or if DGBL (data bus grab signal) is asserted by any peripheral card. If DBGL is used, it must be driven by an *open collector* buffer. When an expansion module is connected, RESETCPUL is held low until the expansion module is powered up. Switching the expansion module off causes RESETCPUL to go low as well.

Appendix G – Error codes

Name	value	type of error
ERR.NC	−1	Not complete
ERR.NJ	−2	Invalid job
ERR.OM	−3	Out of memory
ERR.OR	−4	Out of range
ERR.BO	−5	Buffer full
ERR.NO	−6	Channel not found
ERR.NF	−7	Not found
ERR.EX	−8	Already exists
ERR.IU	−9	In use
ERR.EF	−10	End of file
ERR.DF	−11	Drive full
ERR.BN	−12	Bad name
ERR.TE	−13	Xmit error
ERR.FF	−14	Format failed
ERR.BP	−15	Bad parameter
ERR.FE	−16	Bad medium
ERR.XP	−17	Error in expression
ERR.OV	−18	Overflow
ERR.NI	−19	Not implemented (yet)
ERR.RO	−20	Read only
ERR.BL	−21	Bad line

Appendix H – Trap summary

Of the available traps, 0 to 4 are defined for use by QDOS, the remaining 5 to 15 can be defined for user applications. For more detailed information about traps and their uses, see chapters 5 to 7.

Trap #0

This trap is used to change from user mode to supervisor mode. None of the registers are affected by this trap.

Trap #1 – Manager Traps

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 removes a Job from transient program area
06	MT.FREE	Finds largest contiguous 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 an external interrupt service routine into QDOS
1B	MT.RXINT	Removes an external interrupt service routine
1C	MT.LPOLL	Links a polling 50/60Hz service 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

Trap #2 – IO Allocation

D0	Name	Description
01	IO.OPEN	Opens a channel
02	IO.CLOSE	Closes a channel
03	IO.FORMT	Formats a sectored medium
04	IO.DELET	Deletes a file

Trap #3 – IO Utilisation

D0	Name	Description
00	IO.PEND	Checks for pending input
01	IO.FBYTE	Fetches a byte
02	IO.FLINE	Fetches a line of ASCII characters terminated by <LF>
03	IO.FSTRG	Fetches a string of bytes
04	IO.EDLIN	Edits a line of characters (console driver only)
05	IO.SBYTE	Sends a byte

07	IO.SSTRG	Sends a string of bytes
09	SD.EXTOP	Invokes additional routine as part of screen driver
0A	SD.PXENQ	Returns window size and cursor position (pixels)
0B	SD.CHENQ	Returns window size and cursor position (characters)
0C	SD.BORDR	Sets the border width and colour
0D	SD.WDEF	Redefines a window
0E	SD.CURE	Enables the cursor
0F	SD.CURS	Suppresses the cursor
10	SD.POS	Position cursor at row,column in character intervals
11	SD.TAB	Position cursor at any defined character column
12	SD.NL	Position cursor on a new line of characters
13	SD.PCOL	Position cursor on previous character column
14	SD.NCOL	Position cursor on next character column
15	SD.PROW	Position cursor on previous character row
16	SD.NROW	Position cursor on next character row
17	SD.PIXP	Position cursor using pixel coordinates
18	SD.SCROL	Scrolls all of a window
19	SD.SCRTP	Scrolls the top of a window
1A	SD.SCRBT	Scrolls the bottom of a window
1B	SD.PAN	Pans all of a window
1E	SD.PANLN	Pans cursor line
1F	SD.PANRT	Pans right hand end of cursor line
20	SD.CLEAR	Clears all of a window
21	SD.CLRTP	Clears the top of a window
22	SD.CLRBT	Clears the bottom of a window
23	SD.CLRLN	Clears the cursor line
24	SD.CLRRT	Clears right hand end of cursor line
25	SD.FOUNT	Sets or resets the fount
26	SD.RECOL	Recolours a window
27	SD.SETPA	Sets the paper colour
28	SD.SETST	Sets the strip colour
29	SD.SETIN	Sets the ink colour
2A	SD.SETFL	Sets flashing
2B	SD.SETUL	Sets underscoring
2C	SD.SETMD	Sets the character writing or plotting mode
2D	SD.SETSZ	Sets the character size and spacing
2E	SD.FILL	Fills a rectangular block within window
30	SD.POINT	Plots a point

31	SD.LINE	Plots a line
32	SD.ARC	Plots an arc
33	SD.ELLIPS	Plots an ellipse
34	SD.SCALE	Sets window scale
35	SD.FLOOD	Turns area flood on and off
36	SD.GCUR	Sets graphics cursor position
40	FS.CHECK	Checks all pending operations on a file
41	FS.FLUSH	Flushes buffers for a file
42	FS.POSAB	Positions file pointer at absolute location in file
43	FS.POSRE	Positions file pointer at relative location in file
45	FS.MDINF	Gets information about the storage medium
46	FS.HEADS	Sets the file header
47	FS.HEADR	Reads the file header
48	FS.LOAD	Loads a file into memory
49	FS.SAVE	Saves a file from memory

Trap #4 – Special for Basic

This trap is especially for the Basic Command Interpreter. It makes addresses passed to IO traps relative to A6. It is used to precede the traps #2 and trap #3. Its effect is cancelled by trap #2 or trap #3. For trap #2, A6 is added to A0 on entry. For trap #3, A6 is added to A1 on entry and removed on exit. Trap #4 has no parameters, no error returns and preserves all registers. Note that trap #4 is **not** cancelled by a trap #3 call which returns with the error NO (not open).

Trap #5 – #15

These traps are not used by QDOS and can be allocated to user applications.

Appendix I – Vectored Utilites Summary

Vector	Name	Type	Description
C0	MM.ALCHP	SM	Allocates common heap area
C2	MM.RECHP	SM	Releases common heap space
C4	UT.WINDW	ST	Set up window using supplied name
C6	UT.CON	ST	Set up a console window
C8	UT.SCR	ST	Set up screen window
CA	UT.ERR0	ST	Write error message to channel 0
CC	UT.ERR	ST	Write error message to given channel
CE	UT.MINT	ST	Convert an integer to ASCII
D0	UT.MTEXT	ST	Send message to a channel
D2	UT.LINK	GU	Link an item into a list
D4	UT.UNLNK	GU	Unlink an item from a list
D8	MM.ALLOC	GU	Allocates an area in a heap
DA	MM.LNKFR	GU	Links free space (back) into heap
DC	IO.QSET	GU	Set up a queue
DE	IO.QTEST	GU	Test queue status
E0	IO.QIN	GU	Put a byte into a queue
E2	IO.QOUT	GU	Extract a byte from a queue
E4	IO.QEOF	GU	Put EOF marker into queue
E6	UT.CSTR	BU	Compare two strings
E8	IO.SERQ	SM	Direct queue handling
EA	IO.SERIO	SM	General IO handling
EC	CN.DATE	BU	Get date and time
EE	CN.DAY	BU	Get day of week
F0	CN.FTOD	BU	Convert floating point to ASCII
F2	CN.ITOD	BU	Convert an integer to ASCII
F4	CN.ITOBB	BU	Convert a binary byte to ASCII
F6	CN.ITOBW	BU	Convert a binary word to ASCII
F8	CN.ITOBL	BU	Convert a binary long word to ASCII
FA	CN.ITOHB	BU	Convert a hex byte to ASCII
FC	CN.ITOHW	BU	Convert a hex word to ASCII
FE	CN.ITOHL	BU	Convert a hex long word to ASCII
100	CN.DTOF	BU	Convert ASCII to floating point
102	CN.DTOI	BU	Convert ASCII to an integer
104	CN.BTOIB	BU	Convert ASCII to a binary byte
106	CN.BTOIW	BU	Convert ASCII to a binary word
108	CN.BTOIL	BU	Convert ASCII to a binary long word
10A	CN.HTOIB	BU	Convert ASCII to a hex byte

10C	CN.HTOIW	BU	Convert ASCII to a hex word
10E	CN.HTOIL	BU	Convert ASCII to a hex long word
110	BP.INIT	BP	Basic procedure initialisation
112	CA.GTINT	BP	Get integers (word)
114	CA.GTFP	BP	Get floating points (6 bytes)
116	CA.GTSTR	BP	Get strings
118	CA.GTLIN	BP	Get long integers (long word)
11A	BV.CHRIX	BP	Reserve space on arithmetic stack
11C	RI.EXEC	AR	Executes an arithmetic operation
11E	RI.EXECB	AR	Executes a list of arithmetic operations
120	BP.LET	BP	Return parameter values
122	IO.NAME	GU	Decode a device name
124	MD.READ	MD	Read a sector on a microdrive
126	MD.WRITE	MD	Write a sector on a microdrive
128	MD.VERIN	MD	Verify a sector on a microdrive
12A	MD.SECTR	MD	Read a sector header on a microdrive

KEY to type codes

SM = System management
ST = Simplified traps
GU = General utility
BU = BASIC utility
BP = BASIC procedure support
AR = Arithmetic package operations
MD = Microdrive utilities

Appendix J – Pointers to BASIC variable lists and stacks

```
BV.START    EQU $00   start of pointers
BV.BFBAS    EQU $00   buffer base
BV.BFP      EQU $04   buffer running pointer
BV.TKBAS    EQU $08   token list
BV.TKP      EQU $0C
BV.PFBAS    EQU $10   program file
BV.PFP      EQU $14
BV.NBAS     EQU $18   name table
BV.NTP      EQU $1C
BV.NLBAS    EQU $20   name list
BV.NLP      EQU $24
BV.VVBAS    EQU $28   variable values
BV.VVP      EQU $2C
BV.CHBAS    EQU $30   channel name
BV.CHP      EQU $34
BV.RTBAS    EQU $38   return table
BV.RTP      EQU $3C
BV.LNBAS    EQU $40   line number table
BV.LNP      EQU $44

BV.CHANGE EQU $48   change of direction marker

BV.BTP      EQU $48   backtrack stack during parsing
BV.BTBAS    EQU $4C
BV.TGP      EQU $50   temporary graph stack during parsing
BV.TGBAS    EQU $54
BV.RIP      EQU $58   arithmetic stack
BV.RIBAS    EQU $5C
BV.SSP      EQU $60   system stack
BV.SSBAS    EQU $64

BV.ENDPT    EQU $64   end of pointers

BV.LINUM    EQU $68   current line number (word)
BV.LENGTH   EQU $6A   current length (word)
BV.STMNT    EQU $6C   current statement on line (byte)
BV.CONT     EQU $6D   continue ($80) or stop (0) processing
                      (byte)
```

```
BV.INLIN    EQU $6E  processing in line clause or not (byte)

BV.SING     EQU $6F  single line execution ON ($FF) or
                     OFF (0)
BV.INDEX    EQU $70  name tab row of last inlin lp index
                     read (word)
BV.VVFREE   EQU $72  first free space in vvtable (long)
BV.SSSAV    EQU $76  saved sp for out/mem to go back to
                     (long)

BV.RAND     EQU $80  random number (long)
BV.COMCH    EQU $84  command channel (long)

BV.NXLIN    EQU $88  which line number to start after
                     (word)
BV.NXSTM    EQU $8A  which statement to start after (byte)
BV.COMLN    EQU $8B  command line saved ($FF) or not (0)
                     (byte)
BV.STOPN    EQU $8C  which stop number set (word)
BV.EDIT     EQU $8E  program has been edited ($FF) or not
                     (0) (byte)
BV.BRK      EQU $8F  there has been a break (0) or not
                     ($80) (byte)
BV.UNRVL    EQU $90  need to unravel ($FF) or not (0)
                     (byte)
BV.CNSTM    EQU $91  statement to CONTINUE from
                     (byte)
BV.CNLND    EQU $92  line to CONTINUE from (word)

BV.DALNO    EQU $94  current DATA line number (word)
BV.DASTM    EQU $96  current DATA statement number
                     (byte)
BV.DAITM    EQU $97  next DATA item to read (byte)

BV.CNIND    EQU $98  inline loop index to CONTINUE with
                     (word)
BV.CNINL    EQU $9A  inline loop flag for CONTINUE
                     (byte)

BV.LSANY    EQU $9B  whether checking list ($FF) or not (0)
                     (byte)
BV.LSBEF    EQU $9C  invisible top line (word)
```



```
BV.LSBAS    EQU $9E  bottom line in window (word)
BV.LSAFT    EQU $A0  invisible bottom line (word)
BV.LENLN    EQU $A2  length of window line (word)
BV.MAXLN    EQU $A4  max no. of window lines (word)
BV.TOTLN    EQU $A6  number of window lines so far (word)

BV.AUTO     EQU $AA  whether AUTO/EDIT is on ($FF) or
                     off (0) (byte)
BV.PRINT    EQU $AB  print from prtok ($FF) or leave in
                     buffer (0)
BV.EDLIN    EQU $AC  line number to edit next (word)
BV.EDINC    EQU $AE  increment on edit range (word)

BV.TKPOS    EQU $B0  pos of A4 in tklist on entry to PROC
                     (long)
BV.PTEMP    EQU $B4  temp pointer for GO.PROC (long)
BV.UNDO     EQU $B8  undo rt stack then redo procedure
                     (byte)

BV.ARROW    EQU $B9  down ($FF) or up (01) or no (0) arrow
                     (byte)

BV.LSFIL    EQU $BA  fill window when relisting at least to
                     here

BV.END      EQU $100 top of BV area
```