Assembler Update - 21st November 1984

The Assembler now has an additional option, 'j', which, if selected, will make the output file an executable job. The default extension for the output file is then .exe and it there are no errors in the assembly of the source file you will be prompted for the amount of data space required by the job. If you just press ENTER, the default size is 256 bytes, using the job option removes the necessity to create a job explicitly, as described on page 4 of the User Guide. NOTE - A delete job trap is required in the source file if a job is to terminate correctly.

COMPUTER ONE ASSEMBLER

for the Sinclair QL Computer

VALENTE computación
Santa Engracia, 88 ☎ 445 32 85
28010 MADRID

CONTENTS

Introduction

Chapter 1. Using the assembler.
1.1 Backing up
1.2 Loading
1.3 Operation and multitasking
1.4 Completion
1.5 Using the code produced from the Assembler

Chapter 2. Source code requirements.
2.1 Source line format
2.2 Constants
2.3 Expressions
2.4 Current program position
2.5 Expression operands
2.6 Opcodes
2.7 Instruction arguments
2.8 Symbols
2.9 Labels
2.10 Comments

Chapter 3. Assembler directives.
3.1 Define constant
3.2 Define storage
3.3 Define constant block
3.4 Equate
3.5 Align
3.6 External reference
3.7 External definition

Chapter 4. Error messages.

Chapter 5. Using machine code routines in BASIC and Pascal
5.1 Introduction
5.2 BASIC enhancement
5.3 The initialisation section
5.4 The definition table
5.5 The example source
5.6 Using machine code routines in Pascal
5.7 Pascal source template

Chapter 6. Instruction set summary.
6.1 Opcode extensions
6.2 Instructions
6.3 Instruction arguments
6.4 Addressing modes
6.5 Generics

Chapter 7. Using the Editor.
7.1 Edit file format
7.2 Editing while running the Assembler
7.3 Editing a file
7.3.1 Moving the cursor
7.3.1.1 Paging
7.3.2 Inserting text
7.3.3 Deleting text
7.3.4 Splitting and concatenating lines
7.3.5 Finishing the edit session
7.4 The editor file menu
7.4.1 Load file
7.4.2 Create new file
7.4.3 Read file
7.4.4 Save a copy
7.4.5 Save and continue
7.4.6 Save listing
7.4.7 Remove error text
7.4.8 Save and Kill
7.4.9 Kill
7.5 The edit menu
7.5.1 Inserting a marker
7.5.2 Cut
7.5.3 Copy
7.5.4 Paste
7.5.5 Show buffer
7.6 String Search
7.7 Editor Tift
7.8 Edit Listing Facility

Introduction

The Computer One assembler is an integrated system designed for the development of assembly language programs on the Sinclair QL, comprising a full 68008 assembler and program editor. The assembler accepts standard Motorola 68000 mnemonics, and may be used to assemble from either memory or microdrive files. The supplied editor allows you to build and update your assembler source files, and examine the error messages produced by the assembler.

Both the assembler and the editor are run as jobs, and thus may be operated concurrently under the QL QDOS operating system. This is known as multitasking, and allows the user to have more than one program operating in the computer at once.

About this guide

This user guide provides the information required to edit and assemble programs using the Computer One assembler on the Sinclair QL microcomputer. This guide is not intended to teach you 68000 assembler programming, or to serve as a reference guide to the QDOS operating system. For further information on these, the reader is referred to the various texts recommended at the end of this section.

The early chapters of the guide describe the use of the assembler, its required input syntax, and its output.

The assembler may also be used to write machine code extensions to SUPERBASIC and Computer One Pascal. This is briefly described in Chapter 5 and example routines are provided.

The 68000 instruction set is described in Chapter 6.

Finally, Chapter 7 explains the operation of the supplied multitasking program editor.

Using the Assembler with QDOS Multitasking

Before explaining the operation of the Assembler and editor in detail, it will be useful to have some understanding of how the QDOS multitasking operates, and why it is used.

Multitasking with the assembler and the editor allows both programs to be run concurrently, thus allowing you to switch quickly between the editor and the assembler, and to edit a file while another file is being assembled. By keeping both programs in memory, you can quickly switch between the functions without having to load them from microdrive each time you need them. Later chapters describe how the editor and assembler are loaded as jobs.

With the editor and the assembler running concurrently, you can switch the input or control of the jobs from the keyboard simply by pressing CTRL-C (the CTRL and C keys pressed together). The position of the flashing cursor on the screen will indicate which job is currently receiving input. Pressing CTRL-C repeatedly thus allows you to switch between SUPERBASIC, the assembler, and the editor in sequence (and any other jobs that may be also running).

When editing or assembling very large files, you may also choose to load fewer jobs into memory, giving you more space to edit or assemble the program.

Bibliography

Three books you may find useful are:

68000 Assembly Language Programming - Kane, Hawkins and Leventhal
publishers: Osborne/McGraw-Hill, ISBN 0 931988-62-4
A substantial work on the 68000 and assembly language generally, rather American and very heavy going at times, but an excellent introduction.

MC68000 16-Bit Microprocessor User's Manual - Motorola
publishers: Prentice Hall
The reference work on the processor, with an excellent page by page itemisation of the instructions. Partly hardware oriented, but still the best software reference manual by far.

QL Advanced User Guide Adrian Dickens
publishers: Adder Publishing, Cambridge (available from Computer Mart).
This book describes QDOS, the 68008 and interfacing to BASIC. We have found this book a great help in the writing of our software. We highly recommended it should you wish to use the QL to its full potential.

Chapter 1

Using the assembler

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 dire emergency. Please note that you may only copy the software for your own use.

1.2 Loading and multitasking

The assembler is run as a QDOS job by using the EXEC command. Type

EXEC mdv?_Class (where ? is the drive holding the file)

The microcartridge can be in any drive.

When the assembler has been loaded a new window will appear in the middle of the screen and you will be prompted for a filename. Since the assembler is multitasking (with SuperBASIC and possibly the editor running concurrently) you will have to switch the keyboard input into the assembler job at this stage. CNTL-C is used to switch the input between jobs. Note that if you use the EXEC_W command to load the assembler all other jobs are suspended until it has finished and you will not need to press CNTL-C.

1.3 Operation and Command Options

When you are entering a filename the default drive is mdv2_ and need not be specified. The default file extension for source files is '_asm' and for code files is '_cde' ('_lnk' for linker output) and these need not be specified. These extensions will automatically be appended to the file name entered if not already present.

Examples :

source	- the assembler will attempt to assemble the file mdv2_source_asm.
mdv1_src_abc	- the assembler will attempt to assemble the file mdv1_src_abc_asm

You will next be prompted for the assembler options. There are two options :

F - If this option is selected the assembler will assemble the file as it is read from microdrive, or any other device which you may be assembling from. Otherwise the whole file is loaded and assembled from memory; Assembling from memory is of course considerably faster. The default option is to assemble from memory.

L - If this option is selected the Assembler will output the code in linker format (Sinclair Relocatable Object File Format-SROFF). The default is to produce code which may be LBYTEd into memory for execution or subsequent SEXECing. Note SROFF files are not executable but must be submitted to a linker.

The default options are selected by just pressing ENTER when prompted. To change an option from the default just type the letter representing the option and press ENTER.

Example : Assembler options : f1 (ENTER)
Assemble from file and produce SROFF

If the default option of assembling from memory is chosen and there is insuffient room, the system will issue a warning and will automatically assemble from the file.

If you get an 'out of memory' error while assembling, it means that the assembler is unable to get free memory to expand the symbol table. This is likely to happen if there are other jobs in memory with large space requirements. For example if you are multi-tasking with the editor and have allowed a large workspace there may not be sufficient for the assembler. If this happens then try to run the assembler with less space-demanding jobs or on its own.

When the options have been chosen, you will be prompted for a code file name. If you just press ENTER the name will default to the source file name but with the extension '_cde' ('_lnk' if you have selected the linker option). When you have entered a valid code file name the assembler will attempt to delete the old code file, if it exists. Note that the code file may be on a different device.

You will then be prompted for a listing file name. If you press ENTER no listing file will be produced, otherwise a listing will be output to the specified file. This file may be specified either as a standard microdrive file (e.g. mdv1_fred_lst) or could be another QDOS device name (e.g. ser1) such as a printer.

The source file is then assembled, with any error lines being displayed, along with the error numbers. (Pressing CNTRL and F5 will stop/restart the display). Use the editor to display the source file with the embedded error messages (see 7.8).

1.4 Completion

When the assembly is complete a message is displayed which gives you the option of killing the assembler job (press 'k'), of pressing CNTL-C to switch to another job (this only applies if you did not use the EXEC_W command), or of pressing any other key to assemble another file. If you have pressed CNTL-C, then you will still have to press another key, except 'k', to assemble another file when you have come back to the assembler (by pressing CNTL-C from another job). Note that if you use EXEC_W and then kill the assembler job you will have to wait for 15 seconds before another job can be used. This is because the assembler is suspended before it is deleted to allow all pending I/O to be completed.

1.5 Using the code produced from the Assembler

A code file produced from the assembler may be loaded into memory to be executed from BASIC or Pascal (or indeed any other application). This is described in chapter 5. You may wish however to turn your code into a QDOS job to be EXECed. If you are not familiar with some of the terms in this section you should read the relevent documentation in the Advanced User Guide or the keywords section of the QL User Guide.

In order to make your code into a job proceed as follows -

1. Assemble the source and note the length of code produced.

2. Reserve enough space using RESPR (if you have any jobs running you will need to reboot the system first - this is unfortunately necessary because of the QDOS restrictions on when RESPR may be used).

3. LBYTE the code file into the reserved space

4. SEXEC to a new file from the reserved area specifying the code length and the stack space your job will require.

An example is given here to illustrate the method.

```
EXEC mdv1 class
Press CNTL-C when the Assembler screen appears
Type the source file name, say JOB, and press ENTER
(the assembler will assume MDV2_JOB_ASM)
Press ENTER for default options (assemble from memory, no linker)
Press ENTER for default code file (MDV2_JOB_CDE)
Press ENTER for no listing
Note the code length, say 500 bytes
Press 'k' to kill the assembler
Press CNTL-C to return to BASIC (not necessary as we will reboot
anyway)
Press the RESET switch on the right of the machine
Press the appropriate mode key
Now do the following in BASIC, assuming your job needs 512 bytes
of stack space

R RESPR(500)
LBYTES mdv2_job_cde,R
DELETE mdv2_job_exe
SEXEC mdv2_job_exe,R,500,512

To run your job do either of
EXEC mdv2_job_exe
EXEC_W mdv2_job_exe
```

Chapter 2

Source code requirements

The assembler accepts source code close to Motorola assembly language format; the actual requirements are detailed below. It does not include macros.

A line of assembler source code may be blank, a comment, or a line containing labels and/or opcodes or directives.

2.1 Source line format

A line takes the syntactic form:

```
[label]  opcode    [argument[,argument]]  [comment]
 or
[label]  directive [arguments]            [comment]
```

where square brackets indicate that the item is optional or depends on the particular context.

The rules for this syntax are:

1- The label, where present, MUST start at the beginning of a line.

2- The opcode, or directive, must be preceded by one or more spaces or tabs.

3- The opcode, arguments, and comment in whatever combination, must be separated by spaces or tabs.

4- Arguments are separated by commas.

Note that any part of the source code may be in upper or lower case at any point in the source file. The assembler ignores case at all points, so that "LABEL_NAME" is the same as "label_name", and indeed "LaBeL_nAmE"! All will be treated as the same label.

2.2 Constants

Constant values may be expressed in decimal, hexadecimal or binary. Hexadecimal numbers are preceded by a dollar ($) sign, binary numbers are preceded by a percent (%) sign. A value may also be a series of characters in single or double quotes, for example 'h' or '1234'. These are converted to their ASCII equivalent so that 'h' is equivalent to $68 and '1234' is equivalent to $31323334. Only the rightmost 4 characters are used.

2.3 Expressions

Expressions consist of symbols, constants and operators. Expressions are evaluated left to right with no precedence of operators. All values are 32 bits with overflow ignored. The operators available are :-

+ - (monadic and dyadic) * / ^ (mod) | (bit or) ~ (unary bit not) & (bit and) < (shift left) > (shift right)

Square brackets([ ]) are used in expressions to override operator precedence rules.

Examples :

```
3 + [4*5]     gives 23
3 + 4 * 5     gives 35
mask1 | reg   returns the result of 'or'ing the two values
count < 4     returns the result of shifting value left 4
              places
[8^3+2]*2     returns 8  (  (2+2)*2  )
```

2.4 Current program position

The current internal program-counter position is represented as an asterisk (*).

```
BRA       *-6
```

will cause a branch to the position 6 bytes before the program counter's value at the start of the instruction. Note that this will assemble as a displacement of -8, as the displacement is calculated from the program counter's position after the instruction has been fetched. It is a relative value.

2.5 Expression operands

An expression may be a single operand which has a value and a type. The types supported are

```
data register
address register
special register ( CCR SR USP - see 2.8 )
relative
absolute
register list
```

Only relative and absolute values may be used as the operands of an operator in an expression. Only '+' and '-' allow the use of relative values as operands in the combinations

```
R+A     (-> R)
A+R     (-> R)
R-R     (-> A)
R-A     (-> R)
```

where R denotes a relative symbol (e.g a label) and A denotes an absolute value (i.e. a constant value not dependent on where the program is located, for example 6, 'a').

A register list is a list of data and address registers or register ranges separated by a slash (/). A register range is a pair of registers both the same kind where the second has a higher number than the first. (Of course the registers may also be symbols EQUated to registers). For example

```
entrymask EQU d4-d7/a2/a4-a5

This is the list d4/d5/d6/d7/a2/a4/a5/a6.
```

2.6 Opcodes

Opcodes are the mnemonics used to represent the 68000 instructions. A list is given in chapter 6.

2.7 Instruction Arguments

The arguments of an instruction give it the source and destination of the information, as necessary. They are in Motorola format. See Chapter 6 for further information. Labels may be used for relative branches and program counter relative addressing, and the assembler will make the appropriate relative calculations. See below under Labels.

2.8 Symbols

Symbols are used to represent values, to make the code easily read and easily altered. A symbol starts with a letter and may contain letters, digits and underscores. Symbols longer than 30 characters will be truncated to 30 characters.

Symbols are treated in exactly the same way as the values they represent. With the exception of a label, a symbol MUST be defined (by EQU or XDEF) on a line before its use, otherwise it will be assumed to be a label.

A symbol may be EQUated to an expression, defined by XDEF or is defined by making it a label at the start of a line.

There are several predefined symbols -

```
SP    -     stack pointer ( EQU a7 ) .
CCR   -     condition code register
SR    -     status register
USP   -     user stack pointer
```

These and the register symbols d0 - d7, a0 - a7 should not be used as labels or redefined using EQU.

2.9 Labels

Labels are used to mark a position in the program to be used for loops, subroutine calls, program counter relative addressing, etc. They must start at the beginning of a line. Unlike other symbols they may be referred to in the text before definition.

Where used as an operand in an expression, a label will be assembled as an offset relative to the beginning of the program, unless the context forces it to be program counter relative (as a branch destination say). The type of a label symbol is called relative. That is, it is either relative to the start of the program or to the current location.

Ten local labels, @0 to @9, may be used between ordinary labels. Although these use less symbol table space they can encourage the writing of difficult to read code. A meaningful label name should be used where possible.

2.10 Comments

Comments are pieces of text placed in the source code as information to the writer of the code, or anyone else reading it. They are ignored by the assembler.

Comments may be placed in the text either on separate lines, where the first character on a line should be a semi-colon (;) or asterisk (*), or after an instruction. In this case they are the last item on the line, and should be preceded by a semi-colon.

Chapter 3

Assembler directives

Assembler directives (also called pseudo-ops) are used to provide information to the assembler, so that it will produce a suitable machine code program. Some, such as Define Constant, actually place values within the program. Others, such as Equate, merely provide information about how to handle the following assembly language.

In the following curly brackets mean zero or more. Square brackets mean optional.

3.1 DC - Define constant

```
Syntax :      DC [ .W | .L ]  expr { , expr }
```

Allocates a number of words or long words with the initial values given by the expressions. The expressions may be absolute or relative. A relative value (label) is its offset from the program start. If no .W or .L is given the default is DC.W.

```
Examples :    DC.W 0, 50, $FFFF
              DC.L 3, lab1, lab2, lab3
```

```
Syntax :      DC.B  (expr | string) { , (expr | string) }
```

Defines a number of bytes with initial values given by the expressions or ASCII characters in a string. A string is a sequence of characters enclosed in matching double or single quotes. A quote within the string is written twice.

```
Examples :    DC.B 'This is a string', 24, 34, 'another string'
              DC.B 'You mustn''t do that', 10, "*"
```

3.2 DS - Define storage

```
Syntax :      DS [ .B | .W | .L ] expr
```

Reserves an area of uninitialised memory at the current PC location, the number of bytes, words or longwords being given by the expression. The expression must evaluate to a constant, not a relative value.

```
Examples :    DS.B 20     reserves 20 bytes at current PC location
              DS.L blocklen  reserves blocklen long words
```

3.3 DCB - Define constant block

```
Syntax :      DCB [ .B | .W | .L ]  expr1, expr2
```

Reserves a block of memory at the current program counter, the number of bytes, words or longwords to be reserved being given by the first expression. The whole block is initialised with the value given by the second expression. Note that the first expression must evaluate to a constant.

```
Example : DCB.W 30,1  reserve a block of 30 words initialised to 1
```

3.4 EQU - Equate

```
Syntax : symbol EQU expr
```

Equates the symbol with the value and type of the expression. The expression must evaluate to a constant, relative value, address register, data register or register list. The symbol MUST NOT be referred to before the EQU otherwise it will be made a label and the EQU will give an error.

3.5 ALIGN

```
Syntax :      ALIGN or
              EVEN
```

Ensures that the code following the align is on a word boundary. Note that all longwords and words are automatically aligned.

The following directives are used when you want the assembler to produce linker format. This is invoked by the L option on assembly.

The output is known as Sinclair Relocatable Output File Format, and is standardised for all products, such as compilers, for the QL. The output uses a subset of the directives defined in the (unpublished) Sinclair Research document entitled 'Sinclair relocatable object file format' revision 3 dated 23rd August 1984.

3.6 XREF - External reference

Syntax : XREF symbol [, symbol]

This signifies that a label referenced within the program is defined in another file. The assembler will make a special entry in its output to indicate that the linker should substitute the value.

A linker is a special program designed to take several files, which may be output by assemblers or compilers, and make a single executable code file from them. XREF allows execution to be passed to, or data to be accessed from, one of these other files. Symbols introduced by XREF are relative values similar to ordinary labels. They may not be used to initialise memory. A symbol in the XREF list MUST NOT be referenced in the program before the XREF otherwise it will be assumed to be an ordinary label and the XREF will give an error.

3.7 XDEF - External definition

Syntax : XDEF symbol [, symbol]

Leaves an entry in the output file to indicate to the linker program the position of a label within this file. This allows the linker to pass this information on to any files making an external reference to it. The symbol MUST NOT be referred to before the XDEF.

3.8 Conditional Assembly

```
Syntax :    IF  expr
            ...
            [ ELSE ]
            ...
            ENDIF
```

The one-armed conditional is when the ELSE has been omitted. The code between the IF and the ENDIF is only assembled should the expression evaluate to a non-zero value.
In the two-armed case, the code between IF and ELSE is only assembled should the expression evaluate to a non-zero value. Otherwise (i.e. with a zero valued expression), the code between ELSE and ENDIF is assembled.

N.B. The expression must be of constant type.
Conditionals may not be nested.

Chapter 4

Error messages

The assembler produces the following messages to indicate a wide range of faults in the source file. They all have a short textual message, this section gives further information and possible causes.

There are, fundamentally, two kinds of error. Either an error which stops assembly processing, or an error which produces incorrect object code. The former includes such catastrophes as "out of memory", the latter such mistakes as "Unknown instruction mnemonic". The first is fatal, and will cause processing to cease. The second will be noted as an error, and may be examined by invoking the editor.

1. Unknown instruction mnemonic - the mnemonic has been mistyped or is not a Motorola mnemonic

2. Bad length specifier - should be .b .w .l or .s depending on the context

3. Wrong number of operands - refer to a Motorola instruction set reference text to find out how many arguments the instruction needs

4. Illegal operand[s] - refer to a Motorola instruction set reference text to find out what arguments are legal for the instruction

5. Bytes operation with address register - .b may not be used where a destination is an address register

6. Relative not allowed as operand[s] - a label or other relative type expression may only be used with + or - in certain combinations and not at all with other operators

7. Missing symbol - a bracket or comma or other such symbol has been omitted

8. Illegal operation on relative type - R + R and A - R are meaningless expressions

9. Not allowed as operand - a register list or register may not be an operand of an operator

10. Expression expected

11. Bad digit

12. Bad character

13. Immediate data too large - the value is too big to fit into the length specified or implied by the instruction or directive

14. Bit number out of range - bit numbers must be in either the range 0 - 7 or 0 - 32 depending on the context

15. Symbol already defined - a symbol may be defined once only in an assembly

16. No label on line - the directive expects a label on the line, e.g. the EQU directive

17. Unterminated string - the terminating quote which must match the starting quote of a string, is missing

18. Relative branch out of range - the relative distance between a branch and its destination is too much for the instruction form. If a .s opcode extension has been used then it should be removed

19. Illegal trap number - trap numbers must be in the range 0 - 15

20. Offset exceeds 16 bits - a signed offset is outside the range -32768 - +32767

21. Offset exceeds 8 bits - a signed offset is outside the range -128 - + 127

22. Operand must be absolute - the operand of a monadic operator may not be a relative type

23. Value must be in range 1 to 8

24. Length specifier not allowed here - the directive should not have a .b .w .l or .s extension

25. Expression must be absolute

26. Illegal range - a register list range consists of ra-rb where both ra and rb are the same type of register and the register number of rb is greater than ra

27. Address register expected

28. Register expected - either a data or address register is expected at this point

29. Quick value out of range - moveq values must be in the range -128 - +127, other quick values must be in the range 1-8

30. @0 undeclared

31. @1 undeclared

32. @2 undeclared

33. @3 undeclared

34. @4 undeclared

35. @5 undeclared

36. @6 undeclared

37. @7 undeclared

38. @8 undeclared

39. @9 undeclared

40. Directive must only be used with linker option

41. External symbols must be relative - symbols which are imported or exported must be labels

42. Undefined symbols - symbols have been referred to but not defined

43. Extra symbols on line - spurious text has been left unprocessed on the line. Either a semicolon is missing before a comment or the previous text on the line contained syntax errors, usually a missing comma or operator

44. Bad local label - a local label should be an 'at' sign (@) immediately followed by a single digit

45. Illegal label - a symbol has been used which cannot be a label

46. External not allowed here - an external relative value should only be used as a label

47. Symbol defined after use - only labels defined in the file may be used before definition

48. Conditional may not be nested

49. Missing conditional - an IF or ENDIF has been omitted

Chapter 5

Using the Assembler to enhance SuperBASIC

5.1 Introduction

For many programmers, BASIC and Pascal prove adequate for writing applications programs. However, having written a program in them, you may well wish to make it smaller and/or faster. You may also set out to write a program in a mixture of one of these and Assembler.

The following section sets out to show you how to add procedures and functions to BASIC by using a commented example. IMPORTANT NOTICE. This section sets out only to be a simple guide to adding your own procedures and functions to BASIC. The information in it is by no means complete or definitive for all possible programs. If you wish to write such procedures and functions you require fuller documentation on QDOS, which is available from Computer One.

Should you wish to use machine code subroutines from Computer One Pascal this chapter also contains an example program template to illustrate one way of doing so. As you become more familiar with the additional facilities available in Pascal you may find alternate methods. Computer One Pascal already contains several routines to efficiently implement memory access and data movement.

5.2 BASIC enhancement

Adding assembly language procedures and functions to BASIC is a fairly simple operation. Entries are added to BASIC's internal name table, and the subroutines that are then called appear from BASIC as if they has been included in the ROM.

The QL's operating system, QDOS, is used via a set of TRAPs and VECTORs lying at the bottom of the ROM. These allow you to call parts of QDOS whichever version of it you have. These are the only safe way to use QDOS.

5.3 The initialisation section

The names of the functions are added into BASIC's name table, to initialise them before use, by a short routine; this loads a pointer to a list of definitions into A1, and a vector into A2. It takes the form:

```
        LEA     PROC_DEF,A1             ;pointer to definition table
        MOVE    BP_INIT,A2              ;initialise vector
        JSR     (A2)                    ;do it
        RTS                             ;return to BASIC
```

It is worth noting that the first line references the label PROC_DEF relative to the program counter. This is because, like all QL assembly language programs, this routine is POSITION INDEPENDANT. This means that wherever it is loaded into memory it will operate correctly.

If you try to use a label in a position where it cannot be made program-counter relative, for instance as a destination operand, then the assembler will warn you with an error message.

Then the vector BP_INIT (BASIC Fn/Proc INITialise - $110) is loaded into A2. BP_INIT has been previously EQUated to the value $110. The MOVE instruction will automatically be assembled into a MOVEA instruction. MOVEA defaults to word length, so the word value at the address $110 is sign-extended into it, giving it the long-word value stored as a word at this location.

This is then jumped to as a subroutine, and then finally a return is made to BASIC.

5.4 The definition table

The definition table takes the form:

```
        DC.W    2                       ; number of procedures, 2 here
        DC.W    CON_OF-*                ; word offset to 1st proc
        DC.B    @2-@1                   ; length of name
@1      DC.B    'CURSON'                ; name of first procedure
@2
        ALIGN                           ; make even boundary
        DC.W    COFF_OF-*               ; word offset to 2nd proc
        DC.B    @4-@3                   ; length of name
@3      DC.B    'CURSOF'                ; name
@4
        ALIGN                           ; even boundary
        DC.W    0                       ; end of procedure definitions
        DC.W    1                       ; number of functions, 1 here
        DC.W    CONV_OF-*               ; word length offset to first fn
        DC.B    @6-@5                   ; name length
@5      DC.B    'CONV$'                 ; name
@6      ALIGN                           ; even boundary
        DC.W    0                       ; end of function definitions
```

The table is in two sections, a list of procedure definitions, and a list of function definitions. Each is preceded by a word indicating how much space BASIC will need to add them, and terminated by a word of 0. Function and procedure names should be no more than 7 characters long.

Each entry consists of a word length offset, a byte character count, and the ASCII representation of the name. Each entry should start on a word boundary. A word length offset is easily generated by the assembler with DC.W, which is made program-counter relative when its argument is a label by subtracting the current location counter (*). The table above shows you the assembler syntax.

The table given, used as data for the program above, comes near the start of the program. You may load the code into memory and invoke it by going, for example:

```
100 REMark BASIC loader program
110 A=RESPR(512)
120 LBYTES MOVI_CURSE_CDE,A
130 CALL A
```

This reserves space in the resident procedure area, which is intended for this purpose; loads the executable code into it, and calls it. This should pass execution to the subroutine above which will then allow you to access your functions by inputting PRINT CONV$("ConVert ME TO All ONe Case"), for example.

NOTE: CALL does not work reliably when a substantial BASIC program is present in some QLs, so this setting up should be done by a small BOOT program at start-up, before any large BASIC programs are loaded. Note also that RESPR does not allow you to allocate space if there is a job running.

The above lines of BASIC assume that you have assembled the code from the file CURSE_ASM on some device and directed the code to a file MOVI_CURSE_CDE. They also illustrate how ANY machine code subroutine may be called from BASIC.

5.5 The example source

```
; Machine Code routines to add a cursor on/off facility and case
; converter function to BASIC
; Syntax :  CURSON n            turn on cursor in window #n
;           CURSOF n            turn off cursor in window #n
;           CONV$(string)       returns string all in upper case
;
; Symbol Definitions - must precede use
;
ERROR_BP        EQU     -15     ; QDOS bad parameter
CH_LENCH        EQU     $28
BV_RIP          EQU     $58
BV_CHBAS        EQU     $30
BP_INIT         EQU     $110
CA_GTINT        EQU     $112
CA_GTSTR        EQU     $116
SD_CURE         EQU     $E
SD_CURS         EQU     $F
;
; Initialisation routine
;
ADD_FUN
        LEA     PROC_DEF,A1             ;pointer to definition table
        MOVE    BP_INIT,A2              ;initialise vector
        JSR     (A2)                    ;do it
        RTS                             ;return to BASIC
;
; Procedure definition table
;
PROC_DEF
        DC.W    2                       ; number of procedures, 2 here
        DC.W    CON_OF-*                ; word offset to 1st proc
        DC.B    @2-@1                   ; length of name
@1      DC.B    'CURSON'                ; name of first procedure
@2
        ALIGN                           ; make even boundary
        DC.W    COFF_OF-*               ; word offset to 2nd proc
        DC.B    @4-@3                   ; length of name
@3      DC.B    'CURSOF'                ; name
@4
        ALIGN                           ; even boundary
        DC.W    0                       ; end of procedure definitions
        DC.W    1                       ; number of functions, 1 here
        DC.W    CONV_OF-*               ; word length offset to first fn
        DC.B    @6-@5                   ; name length
@5      DC.B    'CONV$'                 ; name
@6      ALIGN                           ; even boundary
        DC.W    0                       ; end of function definitions
;
;
; CODE FOR CONV$
;
CONV_OF
        move    CA_GTSTR, a2            ; get string argument from BASIC
        jsr     (a2)
        bne.s   EXIT                    ; error return
; ok
        subq    #1, d3                  ; check if one parameter
        bne.s   ERR_BP                  ; error if not
; ok
        move.w  0(a6,a1.l), d7          ; get length of string
        move    a1, a0                  ; take copy of a1
        addq.l  #2, a0                  ; increment pointer
SCAN_START
        tst.w   d7                      ; are we at end of string
        beq.s   SET_STACK               ; leave if so
        subq.w  #1, d7                  ; do another character
        move.b  0(a6,a0.l), d1          ; get char
        cmpi.b  #'a', d1                ; is it lower case
        blo.s   NO_CONV
        cmpi.b  #'z', d1
        bhi.s   NO_CONV
; is lower case,so convert
        subi.b  #'a'-'A', 0(a6,a0.l)
NO_CONV
        addq.l  #1, a0                  ; increment char pointer
        bra.s   SCAN_START
SET_STACK
        moveq   #1, d4                  ; tell BASIC a STRING type has been
                                        ; returned
        move.l  a1, BV_RIP(a6)          ; tell BASIC where its stack top is
        clr.l   d0                      ; no errors
        rts
;
ERR_BP
        moveq   #ERROR_BP, d0           ; bad parameter
EXIT
        rts
;
; CURSOR ON/OFF routines
;
CON_OF
        move.b  #SD_CURE, d5            ; cursor enable
        bra.s   CURS_START
COFF_OF
        move.b  #SD_CURS, d5            ; cursor disable
CURS_START
        moveq   #1, d4                  ; default channel id
        move    CA_GTINT, a2            ; also puts stack ptr into a1
        jsr     (a2)
        bne.s   EXIT                    ; error
        subq.w  #1, d3                  ; check there is one parameter
        bne.s   ERR_BP
        move.w  0(a6,a1.l), d4          ; channel number into d4
        addq.l  #2, a1                  ; unstack it
        move.l  a1, BV_RIP(a6)          ; tell BASIC about stack top
        mulu    #CH_LENCH, d4           ; look up channel table
        add.l   BV_CHBAS(a6), d4        ; d4 now has channel id offset
        move.l  0(a6,d4.l), a0          ; extract channel id
        move.l  #500, d3                ; timeout
        move.b  d5, d0                  ; operation to perform, on/off
        trap    #3                      ; error/ok in d0 on return
        rts
        END                             ; of assembly
```

5.6 Using machine code routines in Pascal

Machine code routines may be called from Computer One Pascal by means of the 'call' procedure. This takes an address, a vector of data register values and a vector of address register values. The types of these are all predefined in Computer One Pascal. The call procedure causes the registers to be loaded with the values in the arrays and execution to commence at the specified address. The machine code routine should, on exit, leave the stack as it found it on entry, just prior to return the values in the registers will be copied into the arrays. This is an extremely simple and easy to use mechanism.

The machine code routine must be loaded into memory before execution and must reside in memory until it no longer is needed. Space must be allocated to hold it. There are three main ways of allocating space

- use an array declared at the outer level
- use an array local to a procedure or function BUT remember this will disappear on exit from the procedure or function
- allocate an array on the heap

The location of an array or variable in Computer One Pascal may be found by using the 'loc' function. This returns a value of type 'address' which may be directly passed to the 'call' procedure. The machine code file may be loaded into memory using the 'lbytes' procedure. (Note that 'lbytes' can be used to load any file into memory, for example a data file or text file).

A section of a Pascal program is given here to demonstrate the technique.

5.7 Pascal source template

```
program mac;
const
   codelength = ... ; { size of file to be lbyted }
...
var
   d : dreg ; { array [0..7] of integer (predeclared type) }
   a : areg ; { array [0..5] of integer (predeclared type)
                { note a6 and a7 are sacrosanct                }
   { now reserve space to hold the machine code }
   code : array [ 1..codelength ] of char; { codelength bytes }
   entry : address ;
...
begin
```

```
    ...
    { load in the code }
    entry := loc( code ) ;
    lbytes( 'mdv._..._cde', entry ) ; { appropriate file name }
    ...
    { now set up the call }
    d[0] := ... ; a[0] := ... ; { only set up registers needed }
    call( entry, d, a ) ; { call the machine code routine }
    ... := d[0] ; ... := a[0] ; { etc as needed }
    ...
end.
```

You may of course wish to have multiple sets of address and data register values, using the appropriate sets on each call.

Chapter 6

Instruction set summary

6.1 Opcode extensions

The length extensions for each instruction can be some or all of .b (byte), .w (word) or .l (long-word). The length entry for each instruction shows the initials of the allowable lengths, where relevant. Where a length is needed by an instruction but the length specifier is missing .w is assumed except where the instruction does not accept this (check with an instruction set reference). Branch instructions may also use the .s (short) extension.

The argument forms are summarized below.

The flags are shown according to the following convention:

```
* = set according to the result of the operation
- = not affected
0 = cleared
1 = set
U = undefined after operation
```

6.2 Instructions

Mnemonic	Length	Argument forms	Flags XNZVC	Operation
ABCD	B	Dn,Dn	*U*U*	Add decimal with extend
	B	-(An),-(An)	*U*U*	
ADD	BWL	<ea>,Dn	*****	Add binary
	BWL	Dn,<ea>	*****	
ADDA	WL	<ea>,An	-----	Add address
ADDI	BWL	#<data>,<ea>	*****	Add immediate
ADDQ	BWL	#<data>,<ea>	*****	Add quick
ADDX	BWL	Dn,Dn	*****	Add extended
	BWL	-(An),-(An)	*****	
AND	BWL	<ea>,Dn	-**00	And logical
	BWL	Dn,<ea>	-**00	
ANDI	BWL	#<data>,<ea>	-**00	And immediate
	B	#<data>,CCR	*****	
	W	#<data>,SR	*****	
ASL/ASR				
	BWL	Dn,Dn	*****	Arithmetic shift
	BWL	#<data>,Dn	*****	left/right
	W	<ea>	*****	
Bcc	S	<label>	-----	Branch conditionally
BCHG	L	Dn,Dn	--*--	Test a bit and change
	L	#<data>,Dn	--*--	
	B	Dn,<ea>	--*--	
	B	#<data>,<ea>	--*--	
BCLR	L	Dn,Dn	--*--	Test a bit and clear
	L	#<data>,Dn	--*--	
	B	Dn,<ea>	--*--	
	B	#<data>,<ea>	--*--	
BRA	S	<label>	-----	Branch always
BSET	L	Dn,Dn	--*--	Test a bit and set
	L	#<data>,Dn	--*--	
	B	Dn,<ea>	--*--	
	B	#<data>,<ea>	--*--	
BSR	S	<label>	-----	Branch to subroutine
BTST	L	Dn,Dn	--*--	Test a bit
	L	#<data>,Dn	--*--	
	B	Dn,<ea>	--*--	
	B	#<data>,<ea>	--*--	
CHK	W	<ea>,Dn	-*UUU	Check register against bounds
CLR	BWL	<ea>	-0100	Clear an argument
CMP	BWL	<ea>,Dn	-****	Compare
CMPA	WL	<ea>,An	-****	Compare address
CMPI	BWL	#<data>,<ea>	-****	Compare immediate
CMPM	BWL	(An)+,(An)+	-****	Compare memory
DBcc	W	Dn,<label>	-----	Test cond, decrement and branch
DIVS	W	<ea>,Dn	-***0	Signed divide
DIVU	W	<ea>,Dn	-***0	Unsigned divide
EOR	BWL	Dn,<ea>	-**00	Exclusive or logical
EORI	BWL	#<data>,<ea>	-**00	Exclusive or immediate
	B	#<data>,CCR	*****	
	W	#<data>,SR	*****	
EXG	L	Rn,Rn	-----	Exchange registers
EXT	WL	Dn	-**00	Sign extend
JMP		<ea>	-----	Jump
JSR		<ea>	-----	Jump to subroutine
LEA	L	<ea>,An	-----	Load effective address
LINK		An,#<disp>	-----	Link and allocate
LSL/LSR				
	BWL	Dn,Dn	***0*	Logical shift
	BWL	#<data>,Dn	***0*	left/right
	W	<ea>	***0*	
MOVE	BWL	<ea>,<ea>	-**00	Move data from source to
	W	<ea>,CCR	*****	destination
	W	<ea>,SR	*****	
	W	SR,<ea>	-----	
	L	<ea>,USP	-----	
	L	USP,<ea>	-----	
MOVEA	WL	<ea>,An	-----	Move to address register
MOVEM	WL	<reg list>,<ea>	-----	Move multiple registers
	WL	<ea>,<reg list>	-----	
MOVEP	WL	Dn,d(An)	-----	Move peripheral data
	WL	d(An),Dn	-----	
MOVEQ	L	#<data>,Dn	-**00	Move quick
MULS	W	<ea>,Dn	-**00	Signed multiply
MULU	W	<ea>,Dn	-**00	Unsigned multiply
NBCD	B	<ea>	*U*U*	Negate decimal with extend
NEG	BWL	<ea>	*****	Negate
NEGX	BWL	<ea>	*****	Negate with extend
NOP			-----	No operation
NOT	BWL	<ea>	-**00	Logical complement
OR	BWL	<ea>,Dn	-**00	Inclusive or logical
	BWL	Dn,<ea>	-**00	
ORI	BWL	#<data>,<ea>	-**00	Inclusive or immediate
	B	#<data>,CCR	*****	
	W	#<data>,SR	*****	
PEA	L	<ea>	-----	Push effective address
RESET			-----	Reset external devices
ROL/ROR				
	BWL	Dn,Dn	-**0*	Rotate (without extend)
	BWL	#<data>,Dn	-**0*	left/right
	W	<ea>	-**0*	
ROXL/ROXR				
	BWL	Dn,Dn	***0*	Rotate with extend
	BWL	#<data>,Dn	***0*	left/right
	W	<ea>	***0*	
RTE			*****	Return from exception
RTR			*****	Return and restore condition codes
RTS			-----	Return from subroutine
SBCD	B	Dn,Dn	*U*U*	Subtract decimal with extend
	B	-(An),-(An)	*U*U*	
Scc	B	<ea>	-----	Set according to condition
STOP		#<word>	*****	Load status register and stop
SUB	BWL	<ea>,Dn	*****	Subtract binary
	BWL	Dn,<ea>	*****	
SUBA	WL	<ea>,An	-----	Subtract address
SUBI	BWL	#<data>,<ea>	*****	Subtract immediate
SUBQ	BWL	#<data>,<ea>	*****	Subtract quick
SUBX	BWL	Dn,Dn	*****	Subtract with extend
	BWL	-(An),-(An)	*****	
SWAP	W	Dn	-**00	Swap register halves
TAS	B	<ea>	-**00	Test and set an argument
TRAP		#<vector>	-----	Trap
TRAPV			-----	Trap on overflow
TST	BWL	<ea>	-**00	Test an argument
UNLK		An	-----	Unlink

6.3 Instruction Arguments

The arguments follow the instruction in the source file. Where two are allowed, they are always in the order <source>,<destination>. Note that for registers and register lists, symbols EQUated to them may be used. For labels, relative expressions may be used. For other data such as offsets and immediate values expressions may be used.

The types shown above are:

Rn	Any address or data register
Dn	Any data register
An	Any address register
<label>	A label or other relative value
<ea>	An effective addressing mode
<data>	Immediate data
<reg list>	A list of registers (see 2.5)
<vector>	Trap number in the range 0-15
<word>	Word to be loaded into the status register
<disp>	A sixteen bit stack pointer displacement
d	A displacement
cc	A condition code - CS, CC, EQ, NE, MI, PL, VS, VC, GT, GE, LT, LE, HI, HS (same as CC), LS, LO (same as CS), T, F

6.4 Addressing modes

NOTE THAT NOT ALL ADDRESSING MODES ARE ALLOWED WHERE <ea> IS SHOWN ABOVE. For further information, a 68000 reference work should be used. Effective addresses are categorised into data, memory, control and alterable modes, or a combination of these. Where you get an illegal operand error it probably means you have used the wrong kind of effective address.

The addressing modes are:

Addressing mode	Syntax
Data register direct	Dn
Address register direct	An
Address register indirect	(An)
Address register indirect with post-increment	(An)+
Address register indirect with pre-decrement	-(An)
Address register indirect with displacement	d(An)
Address register indirect with index and displacement	d(An,Ri)
Program counter relative with index and displacement	label
Program counter relative with index and displacement	label(Ri)
Immediate	#<data>
Absolute address	expression

Zero displacements for d(An) and d(An,Ri) must be present and may not be omitted. In addition to the syntax given under arguments above, note that:

Ri Any register followed by an optional .w or .l, where word length is the default if missing.

Note also that the forms label(PC) and label(PC,Ri) are not available but are equivalent to label and label(Ri) respectively.

6.5 Generics

The assembler accepts generic instructions and assembles the appropriate form. For example add is assembled into adda or addi depending on the arguments. The generics are listed below.

Generic	May be assembled into
add	adda addi
and	andi
cmp	cmpa cmpi cmpm
eor	eori
move	movea
or	ori
sub	suba subi

Chapter 7

The Editor

The multitasking editor, has been designed with ease of use in mind. It makes use of the cursor control keys, situated on either side of the space bar, and the function keys on the left hand side of the QL keyboard. It also provides a menu to allow you to select file saving and loading options easily. The editor is loaded by using the EXEC command from BASIC. This allows the editor to be used while BASIC or other programs are running. Multi-tasking with the editor is described in section 7.2.

7.1 Edit file format

Edit files consist of lines of printable characters, the lines being separated by the line feed character. These files are called text files. Although the editor only allows you to create text files it does not check, when loading a file, that the file only contains printable characters and line feeds. However you are strongly recommended to edit only text files.

Edit lines can only be as long as the edit window. However there is a special character '<<', which, if it occurs as the first character on a line, will be treated as a concatenation character. When a file is loaded from microdrive, any lines longer than the edit window will automatically be split, with the concatenation character being inserted at the split. When a file is saved, a line will be joined to the previous line if the concatenation character appears at the beginning of the line. If this character is deleted it causes a permanent split in the line.

7.2 Editing while running the Assembler

The editor can be used while the assembler is running. Type CNTL-C to get the cursor flashing in the BASIC console then type

EXEC mdv?_editor (where ? is the drive containing it)

You can then edit while the assembly is taking place by pressing CNTL-C. The editor will ask how much workspace is required. You should enter the size of the largest file you wish to edit PLUS how much extra you expect it to grow. If there is not enough contiguous memory free the editor will beep and prompt you again. If it can find it then the four edit windows are displayed.

When you are in the editor and you wish to redisplay the screen, press F4. To get back to the assembler or BASIC (or indeed any other job you may have running) press CNTL-C repeatedly until the flashing cursor appears in the area of the screen it last was for the particular job you wish to return to.

Note that the assembler has less work space if the editor is running.

7.3 Editing a file

When the editor is invoked a new screen appears with four windows; an edit window, in which the editing of text takes place; a prompt window, in which the editor displays prompts and receives input (for example a file name); a help/error window, in which error and help messages are displayed; a lift window - the lift facility is described in section 7.7. A menu will then appear on the screen, allowing you to choose the file you wish to edit or to choose to edit a new file. This menu, called the Editor File Menu, is described in section 7.4. You may wish to move or copy sections of text. This may be achieved using cutting and pasting with the Edit Menu as described in section 7.5.

When a file has been loaded the edit screen will appear with the first few lines of the file. The cursor, a solid green rectangle, will appear over the first character of the file. If a new file is being edited the edit window will be blank, except for the cursor which will be at the top left corner. The cursor represents the current editing position in the file and all editing operations take place at the cursor position. Whether a new file is being edited or an existing file is being edited, the operation of editing the file is exactly the same and is described in the following sections.

7.3.1 Moving the cursor

The cursor can be moved anywhere in the file by using the four cursor control keys which are situated on either side of the space bar on the QL keyboard. Each of the keys moves the cursor one character position in the direction shown on the key. There are several special cases which affect the movement of the cursor :

1) If the cursor is moved right when it is at the end of a line it moves to the start of the next line.

2) If the cursor is moved left when it is at the beginning of a line it moves to the end of the previous line.

3) If the cursor is moved up or down and the previous or next line is shorter than the current one the cursor moves to the end of the new line.

4) If the cursor moves off the top or bottom of the screen the text is moved down or up so that a new line appears at the top or bottom respectively.

5) The cursor remains stationary if any attempt is made to move it off the beginning or end of the file.

7.3.1.1 Paging

If the SHIFT key and the cursor up or cursor down keys are pressed together, the window will scroll down or up by the number of lines that will fit in the window. The cursor is set to the middle of the window.

7.3.2 Inserting text

To insert text simply start typing characters at the keyboard. Each character is inserted at the cursor position, and the character at the cursor position and all characters to the right are shifted right by one character position. When the cursor is the last character on a line and a new line is required press the ENTER key and a blank line will be inserted below the current line, with the cursor at the start of the new line. The editor has an auto indent facility which sets the cursor under the first non-blank character of the previous line when a new line is taken.

Lines can only be as long as the window and any attempt to insert a character which will make the line too long, will cause the QL to emit a beep and the character will not be inserted. However the end of line character can be inserted and if you wish to have the line longer than the edit window, you can continue the line by inserting the special character '<<' at the beginning of the next line. This character is keyed in by pressing the CNTL, SHIFT and X keys together. The editor will automatically remove this character and the preceding end of line character when the file is saved.

7.3.3. Deleting text

The character to the left of the cursor can be deleted by pressing the CNTL key and the cursor left key together. The cursor and all characters to the right are shifted left by one character position. This is the opposite of inserting a character and is useful for correcting typing errors as text is being entered.

The character under the cursor can be deleted by pressing the CNTL key and the cursor right key together. The cursor remains in the same position and all chacters to the right of the cursor are shifted left by one character position.

Note that the last character in a file is a line feed character and cannot be deleted.

If the CNTL, ALT and cursor left(or right) keys are pressed together all characters from the cursor position to the start (or end) of the line are deleted.

7.3.4 Splitting and concatenating lines

A line is split by positioning the cursor at the position in the line where the split is required and pressing ENTER. This causes the character under the cursor and all characters to the right to be inserted below the line.

Two lines are concatenated by deleting the line feed at the end of the first of the two lines. The line feed is always the last character in any line and is represented as a blank. The line feed is deleted in the same way as any other character, using the CNTL and cursor right keys if the cursor is over the line feed, or using the CNTL and cursor left keys if the cursor is at the start of the next line. When the two lines are concatenated all lines below will be scrolled up by one line. If concatenating the two lines will make the resulting line longer than the edit window the QL emits a beep and the line feed character is not deleted. However the two lines can be concatenated when the file is saved, by inserting the concatenation character '<<' at the beginning of the second line. This character is entered by pressing the CNTL, SHIFT and X keys together.

7.3.5 Finishing the edit session

When you have finished editing the file, or wish to save the file and continue editing the same file or another file, the function key F1 can be pressed to select the Editor File Menu which allows the file to be saved by selecting one of the options.

7.4 The editor file menu

This menu appears on the screen when the editor is initially run, or when the function key F1 is pressed. When the menu is displayed it appears with a number of options. Each option is numbered and is selected by pressing the required number or using the cursor up and cursor down keys to step through the options.

The currently selected option is always highlighted. When the required option has been selected the ENTER key should be pressed to use the option. In certain cases some of the options are unavailable, for example the 'save' options cannot be used when no file is currently being edited. When an option is unavailable it is displayed in red (options are normally displayed in white) and cannot be selected using the numeric or cursor keys. You can press the F1 key again if you want to leave the File Menu and return to the edit window. When you are prompted for a file name you can just press ENTER to return to the menu.

Each of the options is described below.

7.4.1 Load file

If this option is selected and a file is currently being edited, it is saved if it is not a new file, otherwise the prompt

save to which file :

appears in the message window and the file will be saved with the entered file name. When the current file has been saved, or no file is currently being edited, the message

load which file :

appears in the message window. The editor then loads the given file and the edit window appears with the first few lines of the file. You can now start editing the file.

If the file name given has the extension '_asm' the editor checks if there is also a file with the same name, but with the extension '_err'. If one exists the editor automatically uses the '_err' file to produce a listing which allows you to see the error messages produced when the specified file was last compiled. The 'Edit listing' facility is described in section 7.8.

7.4.2 Create new file

If no file is currently being edited the edit window is cleared you can start editing a new file. If a file is currently being edited it is saved if it is not a new file, otherwise the prompt

save to which file :

appears in the message window and the file will be saved with the entered file name.

7.4.3 Read file

If this option is selected you will be prompted for a filename. A copy of the file will be inserted at the cursor position.

7.4.4 Save a copy

This option is only available when a file is currently being edited. The message

save to which file :

appears in the message window and the file is saved with the given file name. You can then continue editing the same file. Note that saving a copy with a given file name does not affect the name of the file being edited. i.e. the file being edited still has its original file name.

7.4.5 Save and continue

This option is available only if a file is currently being edited and it does not contain error text. (Use the 'Remove error text' option to delete error text). If the file being edited is a new file, the message

save to which file :

is displayed in the message window and the file is saved with the given name. You can then continue to edit the file.

7.4.6 Save listing

This option is only available if the file has had an error file merged with it to make it a listing.

The listing, the file with the embedded error messages, is saved to a file with the same name as the file being edited, except that the extension '_lis' is used. This option is useful if you wish to print the listing file. When the listing has been saved you can continue to edit the file.

7.4.7 Remove error text

This option is only available if you are editing a listing. It should be used only after you have corrected the errors indicated by the merged error file. When selected the error text is removed and the file no longer is a listing. Removing the error text allows a copy of the file to be saved or the save and continue option to be selected.

7.4.8 Save and kill

This option is only available if a file is currently being edited. If an existing file is being edited the file is saved with the existing file name, otherwise the file is a new file and the message

save to which file :

appears in the message window and the file is then saved with the given file name.

When the file has been saved the editor job is deleted and you should press CNTL C to return to BASIC. Note that the editor will have to be reloaded if you use this option.

7.4.9 Kill

This option deletes the editor job without saving the changes and you should press CNTL-C as needed to return to BASIC. Note the editor will have to be reloaded if this option is chosen. If a file is currently being edited the message

Lose changes ?

is displayed in the message window and only if you enter 'Y' or 'y' will the editor be exited. Otherwise the edit window will be redisplayed and you can continue to edit the file.

7.5 Edit menu

This menu appears when the F5 key is pressed. It has four options

- cut pieces of text from the edit file
- copy pieces of text from the edit file
- paste pieces of text into the edit file
- display the last piece of text cut or copied

To leave the edit menu you press F1. Each of the options are described below. Some options expect a marker to be set. This is described in the following section.

7.5.1 Inserting a marker

A marker is set in the text by pressing the CNTL and F4 keys together. The character at this position is shown in reverse video. Only one position in the file may be marked at one time. The marker will be removed by any action except moving the cursor (either with the cursor keys, paging or the lift). The marked character will only be displayed in inverse video until it moves off the screen.

7.5.2 Cut

This option is only available if a marker has been set. It allows you to remove a piece of text from the edit file and save it in an internal buffer. The contents of this buffer can then be pasted into another part of the file. To cut text, set a marker (CNTL-F4) at the start of the text to be removed. Then place the cursor after the text to be cut. Now press F5 to get the edit menu, select the cut option and press ENTER. The text will be removed from the file but will remain available to be subsequently pasted. The text cut includes the marked character but does not include the character at the cursor.

You will only be allowed to cut if there is sufficient contiguous free memory to act as the internal buffer. Any existing text in the internal buffer will be deleted before the cut.

7.5.3 Copy

This is similar to the cut option except that the text selected is copied into the internal buffer without being removed from the edit file.

You will only be allowed to copy if there is sufficient contiguous free memory to act as the internal buffer. Any existing text in the internal buffer will be deleted before the copy.

7.5.4 Paste

This option is only available if there is some text in the internal buffer, that is, if you have cut or copied text. The cursor should be moved to where you wish the text to be pasted. Press F5 to get the edit menu then select paste and press ENTER. The text in the internal buffer is inserted at the current cursor position unless the resulting file is too big for the workspace in which case an error will occur and the paste will not proceed.

7.5.5 Show buffer

You may examine the contents of the internal buffer by pressing F5 to get the edit menu, selecting the show buffer option and pressing F5. The buffer contents are then displayed.

7.6 String search

The string search is initiated when the function key F2 is pressed. It searches for a given string from the current cursor position. When F2 is pressed the message

search for which string :

is displayed in the message window. You should then enter the required string. If the string is found in the text the cursor is set to the character after the string in the text. If the string is not found the cursor position remains as it was and a beep sounds. If F2 is pressed together with the SHIFT key, a search is made for the string specified in the last search. If, when the prompt for a string is displayed, just the ENTER key is pressed, you will immediately return to the editor.

7.7 Editor lift

The lift is a means of moving through a file quickly and easily. While in normal edit mode the lift arrow moves up and down as the cursor is moved up and down through the text. The top of the lift window represents the top of the file, the bottom of the lift represents the bottom of the file and the lift arrow represents the cursor position in the file.

When the function key F3 is pressed the editor goes into lift mode. Lift mode allows you to move the lift arrow by using the up and down cursor keys (for extra speed use the ALT key and cursor up or down). When F3 is pressed again the editor returns to the normal edit mode with the cursor in the position implied by the lift arrow . This is most useful for long files where paging many times becomes tiresome.

7.8 Edit listing facility

This facility allows Assembly language programs to be edited, while looking at any error messages produced when the program was last assembled. The assembler produces a file with the extension '_err', which contains information about the errors. If you choose to edit an Assembler file (extension _asm) and an '_err' file exists for it, then the editor automatically uses the '_err' file to produce a listing, which appears in the edit window as a normal file. Below each line of the file which had a compilation error, is a copy of the line and the error messages. Each error message is 'bracketed' by the character '$', which cannot be deleted or inserted in the editor by the user.

The file may be edited as normal and when it is saved any parts of the file enclosed in the '$'s are not written to the microdrive file. Thus the error messages are removed when the file is saved. If, when saving the file, the 'save listing' option is chosen (see 7.4.6) the whole file, including the error messages, are saved to a file with the same name as the Assembler file, but with the extension '_lis'.