CONTENTS

1 GETTING STARTED

At the heart of the Sinclair QL there is a member of the Motorola 68000 family of processors: the Motorola 68008. From a software point of view the 68008 is a full 68000 implementation. Its major difference is that the device package is smaller, and only caters for an 8-bit data bus. An effect of this is that the actual throughput of the processor is reduced, due to overheads in memory addressing. This particular detail should not deter the QL assembly language programmer, who still has at his disposal one of the most powerful state-of-the-art 16/32-bit processors currently available. Also, the 68008 only has 20 of its
maximum 32 address lines brought out to its package pins. This means that the addressing range is limited to 1 Megabyte (if you can call 1 Megabyte a 'limitation'!).

This manual describes a fast, powerful editor/assembler package developed specifically for the Sinclair QL. The full screen editor and the 68000 assembler will be discussed only. Detailed descriptions of the 68000 chip and its instructions are not given. For a full treatise of these, as well as the Sinclair QL QDOS multi-tasking environment, refer to the bibliography at the end of this chapter.

1.1 Making a working copy of the Microdrive cartridge

Before doing assembly language work, make a working copy of the Microdrive cartridge. To do this place a blank cartridge in MDV1 and the master cartridge in MDV2. Enter the command:

```
LRUN MDV2_CLONE
```

This will format the new cartridge, and then copy all the programs from the master onto it. Finally, a directory listing will be given of your new copy. Remove the master cartridge from MDV2 and reset the QL. The editor/assembler package will initialize itself and a copyright front page will be displayed. At this point the package is ready and waiting for use.

1.2 The program cycle

If you are familiar with 68000 programming and the Sinclair QL system, you may wish to skip the rest of this chapter and simply go on to see how the editor and the assembler are used. For many people, however, it is useful to see an example. This is what we shall do here.

EDITING A PROGRAM

Figure 1.1 shows the source listing of a very simple executable type program (i.e., a program that may be executed through SuperBASIC's EXEC command). Assuming a working copy of the editor/assembler has been made and that the copyright front page is staring at you in the face, the first thing we need to do is edit the source program on to the Microdrive. Start the editor by using the command:

```
LRUN MDV1_EDITOR
```

Once the editor has loaded and started, you will see a flashing cursor in the top left-hand corner of the display window. Enter the source program shown in Fig.1.1 by simply typing it in. You may use the TABULATE key to space the text out as shown. At the end of each line press ENTER to go on to the next line. Each line may be edited, if you make a mistake, by using the normal SuperBASIC CTRL-arrow keys. If you get really stuck at any stage, refer to Chapter 2.

Having entered the text fully, save the editor text away on to the Microdrive by using the editor CTRL-S command. When the editor asks for the file name, enter:

```
MDV1_PROG
```

The editor will save the text away in the file 'MDV1_PROG_ASM'. The "save' command does not empty the editor buffer and so the editor will be seen to continue. Leave the editor by using the CTRL_F command. If you perform a DIR of the Microdrive at this point you will see your text
file.

ASSEMBLING THE PROGRAM

Now that the program source is on the Microdrive we can assemble it using the assembler. Enter the command:

```
EXEC_W MDV1_ASM_EXEC
```

The assembler will require the answer to three questions, relating to the source file, possible object code file, and possible listing file.

```
*H Job to write a message
;
; Copyright (c) 1984 McGraw-Hill (UK)
;
             ORG           0
;
MYSELF       EQU           -1
MT_FRJOB     EQU           $05
SD_SETSZ     EQU           $2D
UT_SCR       EQU           $C8
UT_MTEXT     EQU           $D0
;
; Header for debuggers etc.
;
             BRA.S         MESSAGE          ;branch to code
             DEFL          0
             DEFW          $4AFB            ;standard header
             DEFW          7
             DEFB          'Message'
             ALIGN

MESSAGE:     LEA           SCR(PC),A1       ;set up a screen
             MOVE.W        UT_SCR,A4
             JSR           (A4)
             BNE.S         SUICIDE          ;check error return
             MOVEQ         #SD_SETSZ,DO     ;set character size
             MOVEQ         #3,D1            ;wide
             MOVEQ         #1,D2            ;tall
             MOVEQ         #-1,D3           ;no timeout
             TRAP          #3
             LEA           HALLO(PC),A1     ;write a message
             MOVE.W        UT_MTEXT,A4
             JSR           (A4)
SUICIDE:     MOVE.L        D0,D3            ;notify any error
             MOVEQ         #MT_FRJOB,D0     ;force remove
             MOVEQ         #MYSELF,D1       ;myself
             TRAP          #1
SCR:         DEFB          $FF,$04          ;chkbrd/4-pix. border
             DEFB          $04,$00          ;green paper, black text
             DEFW          200,35           ;200x35 pixel window
             DEFW          156,100          ;in the middle
HALLO:       DEFW          5
             DEFB          "Hallo"

;
END
```

Figure 1.1 Simple message writing routine

Enter replies to these questions so as to get the following screen display:

```
Source dev_file  (ASM ) ... mdv1_prog
Object dev_file  (CODE) ... mdv1_prog
Listing dev_file (    ) ...
```

This will assemble the file 'PROG_ASM' on 'MDV1_', and produce the binary object file 'PROG_CODE' on 'MDV1_'. If the source text was edited correctly you will get no error messages and the reports:

```
pass 1
pass 2
Error(s) detected: 0000
Symbol bytes free: 3F4C
Program bytes: OOO4E
Assembler finished
```

will be displayed. If you perform a DIR at this stage you will see that there are now two of your files on the Microdrive.

CREATING AND RUNNING THE EXECUTABLE FILE

Your 68000 program is to run as an executable file (i.e., a job), and it will be necessary, therefore, to load the pure binary file ('PROG_CODE') into memory and then re-save it using the SEXEC statement. The following SuperBASIC statements will create an executable copy on a Microdrive:

```
100 base=RESPR(128)
110 LBYTES mdv1_prog_code,base
120 SEXEC mdvl_prog exec,base,78,128
```

The program could thereafter be executed as a job merely by using the SuperBASIC command:

```
EXEC mdv1_prog_exec
```

Note that the extension ' _EXEC' is not compulsory for executable files, it merely helps you to identify easily those programs that are of an 'executable' nature.

Bibliography

MOTOROLA: 'MC68000 16-Bit Microprocessor User's Manual', MOTOROLA INC., Fourth Edition.

Kane,G., Hawkins,D., and Leventhal,L.: ‘68000 Assembly Language Programming’, Osborne/McGraw-Hill, 1981.

Opie,C.: 'QL Assembly Language Programming’, McGraw-Hill(UK), 1984.

2 THE EDITOR

The full screen editor is simple to operate and yet powerful enough to enable assembler source code to be quickly and efficiently edited. In practice it is important to use this editor and not, for example, the word processor package 'Quill', because the latter does not produce pure ASCII text files on the Microdrives (easily). Pure text files are the only type of file that the complementary assembler can parse.

The editor is designed specifically for the creation of source (textual) programs. It allows up to 400 lines to be edited at any one time, with 72 characters per line. This is more than adequate for two major reasons. First, the size of program developed, at least in the early stages, is not likely to exceed this length, and second, the assembler will permit the inclusion of external 'library' files. If a large program is to be developed it is a simple case of creating one central program that will include as many external source file modules as it takes to produce the entire code.

To invoke the editor, place the Microdrive with the working copy of the assembler package (see Chapter 1) into drive 1, and reset the QL. After the normal copyright front page has been displayed, enter:

LRUN MDV1_EDITOR

The editor will be automatically run after being loaded. Note that if the assembler had been used immediately prior to the editor, there would
be no need to reset the QL before entering the above command.

2.1 Editor windows

On entry, the editor screen will appear as shown in Fig.2.1. Four windows exist in the display. Going from the top of the screen to the bottom the function of the windows is as follows.

At the very top there is a ‘status' window. The contents of this window will show, in a continuous manner, the cursor line position,’ the total number of lines used, and the line position of a special marker that can be employed during an edit session.

Below the status window there is a ‘tabulation ruler'. This particular window will never change. The tabulation ruler shows that nine tab-stops are available and that these exist at every eighth column position along a line. Each time the TABULATE key on the keyboard is pressed, the text cursor will move across to the next tab-stop. These tabulation positions are important in that the assembler, described in the next chapter, will tabulate its list-file in an identical way.

The third window in the display is the actual 'text window'. This is truly a window! It is a window with a view into your text. At any one time you can look at up to 17 lines of your program. The flashing cursor within the window enables you to edit your text program easily. When the editor is initially entered the text cursor will be in the top left-hand corner of the text window, corresponding to column 1, line 0.

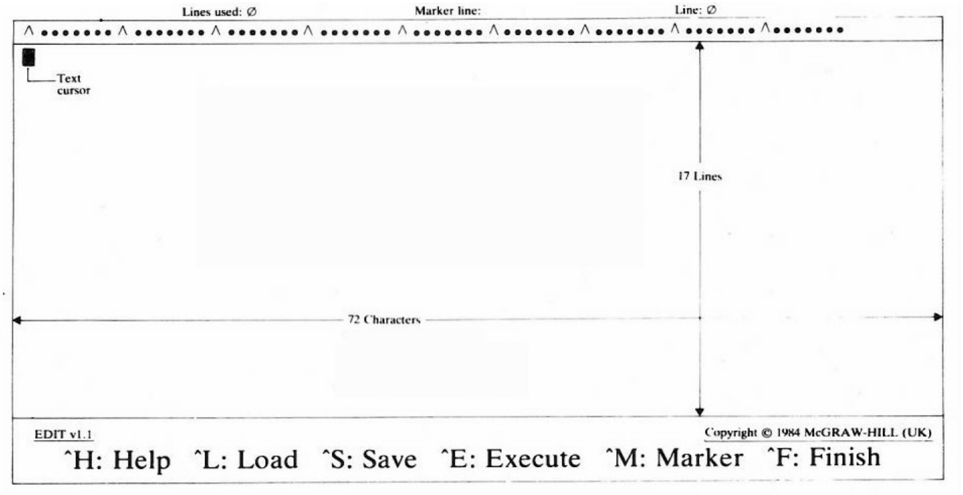

Figure 2.1 Editor screen layout

The last window, at the very bottom of the screen, is used as a message and prompt window. There will be a number of occasions when the contents of this window change. For example, when you request help, the "help' message will appear in this bottom window.

2.2 Editor modes

When the editor is initially entered it will be in its command/edit mode. In this mode there are two major operations available. First, one of six top-level commands (listed in the bottom message window) may be executed. They are all control commands and are therefore entered by typing the appropriate control character (to do this hold the CTRL key down while typing the desired command character). Second, text may be entered and edited simply by typing the required characters, cursor control commands, and text deletion commands.

At first sight it may seem that there are a lot of options to learn. In practice, however, the editor is very simple to use and you can obtain a useful 'help' message, which lists all the cursor control and text deletion commands, should you need it!

2.3 Getting help

The editor has three basic groups of commands and a facility exists to enable you to view a list of the commands in each group, together with their function. The three groups are:

1. Top-level commands (e.g., load-file)
2. Cursor control commands (e.g., cursor-down-a-line)
3. Text deletion commands (e.g., delete-character-left)

The message window at the bottom of the screen normally displays the six top-level commands. One of these commands is the ‘help’ command, entered by typing “H (short for CTRL-H). This will produce a 'help' message in the bottom window showing the 12 cursor control commands, and the four text deletion commands that are available.

Typing a second *H will cause the original message display to be regained. In this way, one control command is used to toggle between two command group lists. Note that this means the editor is totally self-documenting in terms of its command availability.

2.4 Entering text

Text is entered simply by typing in the characters required. On any one line a maximum of 72 characters may be entered. To move on to the next line press the ENTER key, and then continue as before. At any time, except at the very end of a line, the TABULATE key can be pressed, and the cursor will move to the next available tab-stop on the current line.

The real power of the editor is, of course, in the ability to change text, either because it is wrong or because you wish to delete some lines or add extra lines. To do this we need to be able to move the cursor to the appropriate place in the text, and then delete or enter text accordingly.

2.5 Moving the cursor

A total of 12 ‘immediate’ cursor control commands exist. They are entered by using one of the four cursor control keys (up, down, left, or right) in one of three ways:

1. **NORMAL** - the keys are used on their own.
2. **SHIFT** - the keys are entered as 'shift' keys (i.e., the SHIFT key is held down while the cursor control key is pressed).
3. **ALTMODE** - the keys are entered as 'altmode' keys (i.e., the ALT key is held down while the cursor control key is pressed).

The cursor may be moved left or right along a line in a variety of ways. The cursor may also be moved up and down the text. You will notice if you type in more than eight lines that the cursor will stay in the middle of the text window and the text will scroll around it. If, at some later stage, you position the cursor within the first eight lines you will again notice that the cursor moves up and down and the text stays still. This cursor operation is purpose designed to enable you to see the current cursor line in its true context. This in turn makes editing the text much easier. The function of each of the cursor control commands is as follows:

1. **UP** - The cursor will move up one line. If the cursor is at the beginning of the text no action will be taken. If the line moved to is shorter than the current line, the cursor will be positioned at the end of the new line.
2. **DOWN** - The cursor will move down one line. If the cursor is at the end of the text no action will be taken. If the line moved to is shorter than the current line, the cursor will be positioned at the end of the new line.
3. **LEFT** - The cursor will move one character to the left. If the cursor is at the beginning of a line no action will be taken.
4. **RIGHT** - The cursor will move one character to the right. If the cursor is at the end of a line no action will be taken.
5. **SHIFT-UP** - The cursor will move up one page, equivalent to 16 lines. Notice that, as the text window is 17 lines deep, there will always be an overlap of one line. This feature will help you to scan the text more easily. The cursor will always be positioned at the beginning of the new line.
6. **SHIFT-DOWN** - -The cursor will move down one page, “equivalent to 16 lines. Notice that, as the text window is 17 lines deep, there will always be an overlap of one line. This feature will help you to scan the text more easily. The cursor will always be positioned at the beginning of the new line.
7. **SHIFT-LEFT** - The cursor will move one word to the left. If the cursor is at the beginning of a line no action will be taken.
8. **SHIFT-RIGHT** - The cursor will move one word to the right. If the cursor is at the end of a line no action will be taken.
9. **ALTMODE-UP** - The cursor will move to the beginning of the text.
10. **ALTMODE-DOWN** - The cursor will move to the end of the text.
11. **ALTMODE-LEFT** - The cursor will move to the beginning of the current line.
12. **ALTMODE-RIGHT** - The cursor will move to the end of the current line.

If the cursor is moved to a position within a line, and then text entered in the usual way, the characters will be inserted immediately prior to the character under the cursor. The rest of the line will be seen to pan to the right.

2.6 Deleting text

Only four immediate text deletion commands exist, and these in turn only provide three functions (because two of the commands perform the same task). The commands are entered by using the normal cursor control keys (up, down, left, and right) together with the CTRL key. The function of the commands are as follows:

1. **CTRL-UP**	- The current cursor line will be deleted. The command will not be executed if a marker line exists.
2. **CTRL-DOWN**	- (Same as CTRL-UP).
3. **CTRL-LEFT**	- The character immediately to the left of the cursor will be deleted. The rest of the line will pan to the left. No action will be taken if the cursor is at the beginning of a line. If the character to be deleted is a space (single or as part of a tabulation) then spaces will continue to be deleted until the entire gap to the left of the cursor is erased.
4, **CTRL-RIGHT**	- The character under the cursor will be deleted. The rest of the line will pan to the left. No action will be taken if the cursor is at the end of a line. If the character to be deleted is a space (single or as part of a tabulation) then spaces will continue to be deleted until the entire gap to the right of the cursor is erased.

The above commands enable local text to be deleted. It is often useful to delete whole blocks of program text, and this can be done by using one of the editor ‘execute’ command options.

2.7 The ENTER key

When you have entered a _ reasonable number of program lines into the editor and moved the cursor around the text, you will undoubtedly notice that the ENTER key performs different functions at different times. Its functions may be defined as follows.

If the cursor is at the very end of the text then the ENTER key will move the cursor to the beginning of a newly created line, directly after the previous line. This makes the initial entry of text, and the appending of text, very simple. If the cursor is at the beginning of a line, but that line is not the last line of the text, then the ENTER key will create a new blank line at that point and move the rest of the text down. This enables new lines to be inserted within some current text very easily. If the cursor is within a line then the ENTER key will simply move the cursor down to the beginning of the next line.

2.8 Editor ‘execute’ command options

These commands are entered initially by using the top-level command ^E (CTRL-E). Four commands are available and they will be listed in the bottom message display window when the “E command is given. To execute any one of the options simply press the character key corresponding to the first letter of the option. For example, to execute the ‘find string' option you would simply enter F. The operation of the commands is as follows.

FIND STRING COMMAND

A prompt will be given in the bottom window asking for the textual string to search for. Simply type in your search string and press the ENTER key. The editor will search for the string, beginning at the start of the current cursor line. If the string is found then the cursor will be moved to the beginning of the string, and the text window updated accordingly.

If the search string is not within the text searched, the cursor will remain in its previous position. Note that the case of the text is not relevant. For example, 'This' is exactly the same as 'this', as far as the editor search option is concerned.

It may be that the editor search option places the cursor at a match, which is not the particular one you were looking for. If this is so then remember to move the cursor down to the beginning of the next line, or the search option will simply find the same one again.

DELETE BLOCK COMMAND

This is the one command that requires the use of the special marker, so let us look at this first. A marker symbol (shown as a right-sided comilla - double angled bracket character) can be entered, by using the ^M (CTRL-M) top-level command, to mark any particular line in the text. The marker will always be entered and shown at the beginning of the current line, regardless of the current cursor position, and the cursor moved to the first character in the line. It is not possible to mark a line that is completely full, and neither is it possible to mark more than one line.

Assuming a marked line is available, the 'delete-block' command will delete all lines between the marked line and the current cursor line inclusive. : The command will issue an error message if no marker is present (type any character to continue after the error message is printed).

MOVE TO LINE COMMAND

This command lets you move to an absolute line within the text. A prompt will be given requesting the number of the line to which you wish to move. Enter the appropriate number and press the ENTER key. The cursor will be moved to the beginning of the corresponding line and the text window updated.

If a line number less than zero is entered, the cursor will be moved to the beginning of the text. Conversely, if a line number greater than the total number of lines available is entered, the cursor will be moved to the end of the text.

INCLUDE FILE COMMAND

When editing program text it is very useful to be able to merge in other bits of program text from another file. This command will enable a text file (produced by this editor) to be included in the current source text at the current cursor position. A prompt will be given for the device and file name of the external file, which must be on a Microdrive. Simply enter the appropriate information and the file will be included in the current text. While the text inclusion is taking place a_ series of '+' markers will be displayed in the bottom window to act as an indicator. Without such an indicator the editor could appear to lock up, whereas in fact it is simply doing some internal shuffling.

Care must be taken over this operation. If the requested file does not exist, the editor will report a fatal error and cease running! You are advised to save a copy of your current text on a Microdrive before executing this command. The command will issue an error message if a marker line is present (type any character to continue after the error message is printed). An error message will also be given if the editor runs out of storage in the process of trying to merge in the external file.

When specifying the device and file name of the external file, the extension may or may not be given. If it is left off, the default extension '_ASM' will be used.

2.9 Loading a text file

A new source file can be loaded into the editor from a Microdrive by using the ^L (CTRL-L) top-level command. The file must have been created previously using the editor. Any current text will be erased from the memory of the editor and the cursor will be returned to the beginning of the new text.

When specifying the device and file name of the file, the extension may or may not be given. If it is left off, the default extension ‘ASM’ will be used.

2.10 Saving the current text

The current contents of the editor buffer can be saved on to a Microdrive by using the “S (CTRL-S) top-level command. The contents of the editor will not be erased and therefore the 'save' command can be used any number of times during an editing session for safety backup purposes.

When specifying the device and file name of the file, the extension may or may not be given. If it is left off, the default extension ‘ASM’ will be used. It is not possible to save text that has a marker line in it, and under such a condition an error message will be issued (type any character to continue after the error message is printed).

2.11 Leaving the editor

To leave the editor use the top-level command “F (CTRL-F). You will be asked if you are sure about the option! If you press 'Y', the editor will delete itself and your text will be lost. It is important, therefore, to make sure you have saved (*S command) the current text before finally leaving the editor. Pressing any other key will return you to the editor with the text left intact.

RESTARTING THE EDITOR

If, for any reason, you are returned to BASIC while

using the EDITOR, press:

CTRL C

then type,

GOTO 1050 and press RETURN

3 THE 68000 ASSEMBLER

The 68000 assembler described here is a full implementation, written in machine code for the fast assembly of 68000 programs. It is purpose designed for use with the QL device drivers and will therefore work with any peripheral device attached to the QL (e.g., Microdrives, floppy disks, hard disks, serial and parallel printer interfaces, and so on).

Its specification includes:

1. full 2-pass assembly
2. output streaming to screen, printer or mass storage medium
3. pseudo-operations (e.g., ORG, COND)
4. assembler directives (e.g., *HEADING)
5. simple expression parsing
6. long label names and local labels
7. alternative mnemonics, and
8. external library file inclusion.

Note that this chapter describes the facilities available within the assembler only. It does not attempt to discuss 68000 instructions.

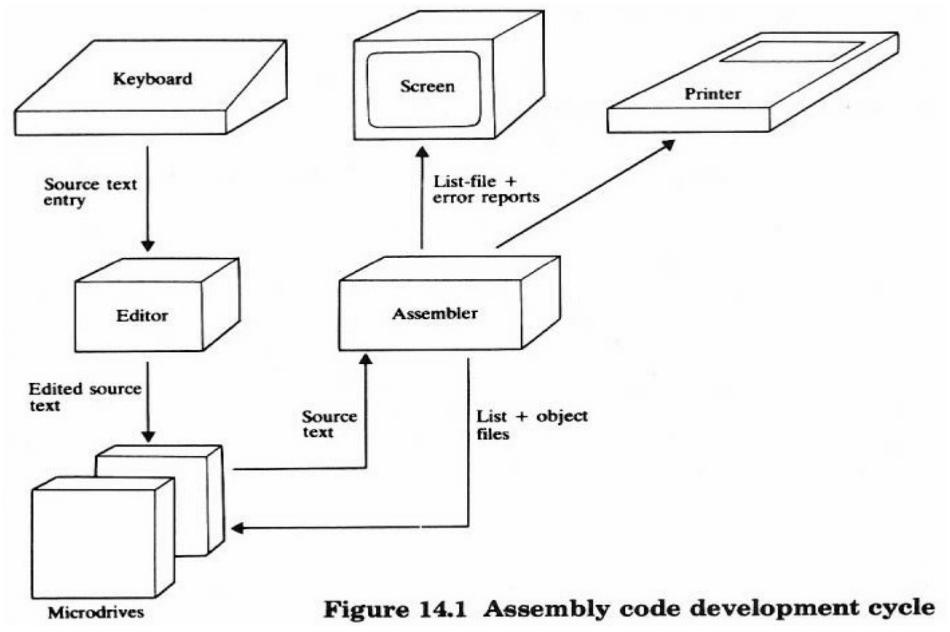

Figure 3.1 Assembly code development cycle using Microdrives

3.1 Assembler operation

The assembler lies at the heart of the assembly language system. It takes its input from a Microdrive file (or some other suitable mass storage medium), and can direct its output either to the screen, a printer, or the mass storage medium. For the purposes of this manual it will be assumed that Microdrives are being used as the mass storage medium. Figure 3.1 illustrates the development cycle. The editor is used first in order to create the source program. This source is then fed to the assembler which creates its various output files. These output files, and in particular the object (binary) file, can then be manipulated in a number of ways. For example, the binary file may be left as it is and accessed by SuperBASIC's LBYTES command. Alternatively its contents could be loaded into memory and then re-saved in the form of an executable file for use with SuperBASIC's EXEC command.

INVOKING THE ASSEMBLER

To invoke the assembler, place the Microdrive with the working copy of the assembler package (see Chapter 1) into drive 1, and reset the QL. After the normal copyright front page has been displayed, enter:

```
EXEC_W MDV1_ASM_EXEC
```

The assembler will be loaded and executed. Note that if the editor had been used immediately prior to the assembler, there would be no need to reset the QL before entering the above command. The '_W' form of the EXEC command must be used in order to suspend SuperBASIC until the assembler has finished.

ASSEMBLER FILE CHANNEL DEFINITIONS

Once the assembler has been invoked it will ask in turn for three file
channel definitions:

1. **Source dev_file (ASM) ...**
   Enter the device and the file name for the source text of your 68000 program. DO NOT enter the extension ' ASM' as the assembler will do this for you (your source program must have an '_ASM' extension as part of its file name). Any device which supports a directory (e.g., Microdrives, floppy disks) may be specified.
   If you simply press ENTER at this stage, without any preceding characters, the assembler will supply you with a short 'help' message and abort.

2. **Object dev_ file (CODE) ...**
   Enter the device and the file name for the binary object code of your 68000 program. DO NOT enter the extension '_CODE' as the assembler will do this for you (an object file will always have a '_CODE'

extension as part of its file name). Any device that supports a directory (e.g., Microdrives, floppy disks) may be specified.

If you simply press EN’ at this stage, without any preceding characters, the assembler will suppress (i.e., not create) any binary object output.

3. **Listing dev_file ( ) ...**
   Enter the device and, if the device requires one, the file name for the assembler listing file. Any extension desired should also be given when appropriate. The assembler will neither put nor assume any extension for the listing file name. Any suitable output device may be specified.

   To send the list file to the assembler's own window you may specify
   the list device as either 'CON_' or 'SCR_'. All four characters must
   be entered. It is not possible to use (i.e., specify) any other
   window.

   If you simply press ENTER at this stage, without any preceding characters, the assembler will suppress (i.e., not create) the assembler listing file.

EXAMPLES

To see the effect of entering various file channel definitions let us consider some examples.

1. **Source dev_file (ASM) ... mdv2_message**
   Object dev_file (CODE) .
   Listing dev_file ( ) pas

   This will assemble the file 'MESSAGE_ASM' on 'MDV2_', but produce no output (other than error messages). This form is therefore useful for performing a quick check on a program to see if it is syntactically correct.

2. **Source dev_file (ASM) ... mdv2_message**
   Object dev_file (CODE) ... mdv2_message
   Listing dev_file ( ) ...

   This will assemble the file 'MESSAGE_ASM' on 'MDV2_', and produce the binary object file 'MESSAGE_CODE' on the same drive. This form will probably be the most widely used one during program creation and testing.

3. **Source .dev_file (ASM) ... mdv2_message**
   Object dev_file (CODE) ...
   Listing dev file ( ) ... ser2c

   This will assemble the file 'MESSAGE_ASM' on 'MDV2_', and send the listing file (during pass 2) out to a printer or some other suitable device attached to 'SER2'. The 'C' postfix merely changes line_feeds to carriage_returns and may not be used for some devices (see QL User Guide). This mode is useful for obtaining hard copy of an assembled program,

4. **Source dev_file (ASM) ... mdv2_message**
   Object dev_file (CODE) ... mdv2_message
   Listing dev_file ( ) ... con_

   This will assemble the file 'MESSAGE_ASM' on 'MDV2_', produce the binary object file 'MESSAGE_CODE' on 'MDV1"', and send the listing file (during pass 2) to the assemblers default window.

ASSEMBLER REPORTS

At the end of a complete assembly operation the following four messages will be given:

```
Error(s) detected: 0000
Symbol bytes free: 3F4C
Program bytes: O004E
Assembler finished
```

though, of course, the actual values displayed will vary. The number of errors detected are displayed as a denary number. The remaining two numerical reports give the value in hex.

PURE BINARY FILES

The assembler always produces a pure binary object file with the extension '_CODE'. If your program is, for example, an extension to SuperBASIC, or a short patch to be called via SuperBASIC's CALL command, this type of binary file is all that you will need.

At the end of assembly the assembler will tell you how long (hex notation) your program was. If you add to this the amount of run-time stack/data space the code will need, you will know how much RAM to reserve for the program using SuperBASIC's RESPR function.

Let us take, for example, the final reports shown above. The program has a length of:

4E (hex) bytes = 78 bytes.

Suppose, on viewing the system calls used etc., that a stack/data workspace of about 100 bytes would be suitable. This means that we must allocate at least '78+100 = 178' bytes for the program. If the SuperBASIC statements:

```
          100 base=RESPR(256)
          110 LBYTES mdv2_prog_code,base
```

were used (assuming the program was called 'PROG' and that it was on "MDV2_'), more than enough space will be allocated. It is certainly better to be safe than sorry!

EXECUTABLE FILES

If your 68000 program is to run as an executable file (i.e., a job), it will be necessary to load the pure binary file ('_CODE') into memory and then re-save it using the SEXEC statement.

If we take our previous example once again, assuming all the values to be the same, the following SuperBASIC statements will create an executable copy on a Microdrive:

```
          100 base=RESPR(128)
          110 LBYTES mdv2_prog_code, base
          120 SEXEC mdv2_prog_ exec, base, 78,100
```

The program could thereafter be executed as a job merely by using the SuperBASIC command:

```
          EXEC mdv2_prog_exec or
          EXEC_W mdv2_prog_ exec
```

Note that the extension '_EXEC' is not compulsory for executable files, it merely helps you to identify easily those programs which are
of an ‘executable’ nature.

3.2 Assembler line syntax

The source input lines for the assembler are single statement lines.

Given here is the general syntax of these lines, more detailed explanations being given later under the appropriate headings.

Assembler source input consists of a series of text lines of maximum length 80 characters, created by the editor described in the previous chapter. Each line is of the form:

LABEL: OPERATOR ARGUMENT ;COMMENT

Any of the four parts - label, operator, argument, or comment - may be omitted where this is appropriate (clearly a blank line would contain none of these, and a pure comment line would contain just the fourth element). Items are separated by one or more blanks (spaces or tab characters), the colon following a label, or the semi-colon preceding the comment.

LABELS

Each label name must start with a letter but thereafter may contain any combination of characters, underscores, or digits. No account is taken of case, everything of importance being converted into upper-case internally. Additionally a temporary label may be given (see Sec.3.4).

OPERATORS AND ARGUMENTS

Operators can be 68000 mnemonics (e.g., ADDX, ROR), assembler pseudo-operators (e.g., DEFB, COND), or an assembler directive (e.g., *INCLUDE). The format of the argument parameter will depend upon the operator that precedes it.

COMMENTS

Any line may have a comment appended to aid source documentation. A comment must be preceded by a semi-colon (;). Anything after this comment delimiter will be ignored by the assembler.

THE ‘END’ PSEUDO-OPERATOR

Assembler source text can optionally be terminated with the END assembler pseudo-operator. If it is not used then the natural end-of-file will be taken as the end of the source text.

3.3. Symbols

Symbols, acting as constants for the duration of the assembly operation,
can be defined either from within the source, or dynamically as boolean
(true/false) constants at assembly time. .

DEFINITION FROM SOURCE (EQU)

Alphanumeric symbols may be defined using the assembler pseudo-operator EQU (or simply an ‘=' sign):

For example:
```
LETA EQU     $41    ;'a'
LETB =       LETA+1 ;'B'
```

The argument following the EQU can be any valid simple expression (as defined later). If an attempt is made to redefine a symbol, an assembler

'M' (Multiple definition) error will ensue - during pass 1 only. If such an error occurs it would be sensible to halt assembly by pressing the ESC key as there may be many future errors, particularly if temporary labels are also being used (which will normally be the case). Upper and lower case are treated as being the same within symbol definitions:

For example:
```
LETC  EQU letb+1    ;'C'
letd  EQU letc+1    ;'D'
LETE  EQU LETD+1    ;'E'
```

Symbols are distinct only within the first eight alphanumeric characters and they must start with an alpha character (A..Z, a..z). If the latter rule is violated an 'L' (Label format) error will ensue.

For example:
```
DELAYforTimer1 = 64
Timer2Delay    = DELAYfor sh1 2
```

DEFINITION AT ASSEMBLY TIME (QRY)

If a symbol is defined with the QRY pseudo-operator, the value may be given as either zero (false) by entering N at the keyboard, or as minus one (true) by entering Y. The prompt for the keyboard entry is given at assembly time (during pass 1), as defined by the QRY argument. For example:

```
FLIST QRY Full listing required
```

will prompt with ‘Full listing required?’ and expect either a Y or an N as the response. The keyboard entry is immediate (no ENTER required) and the assembler will echo either Y or N as appropriate. Note that keying any letter other than Y will effect an N response. This facility is extremely useful when conditional assembly is being used as it allows the programmer to specify flag values at assembly time, and therefore the source does not have to be edited.

3.4 Labels

There are two types of label which can be used. Alphanumeric labels may be defined which will have a scope of the entire program. Temporary or local numeric labels may also be defined, which will have a_ scope limited to the area between the two standard labels in which they are defined.

STANDARD LABELS

A normal alphanumeric label is a special kind of symbol. It is declared by ending it with a colon (:), and it will be given the value of the location counter for the current statement. The label itself must obey the same rules as for symbols (i.e., must be alphanumeric, must start with an alpha character, and be significant in its first eight characters).

TEMPORARY (LOCAL) LABELS

Temporary or local variables have a number of important attributes. Each label takes up only one third of the symbol table space required for normal symbols. They do not appear in the symbol table and therefore the table will refer only to important locations, and they may be re-used within different scope blocks thereby greatly reducing the possibility of multi-defined labels.

A local label is defined by the label form '1%' to '255%' and may optionally be followed by a colon (:). A local label may only exist after a normal label has been declared, and its scope of existence is limited up to the next normal label:

```
nlab1:      moveq #0,d0
            moveq #delay,d1
1%:         cmp.b 4d0,d1
            beq.s 2%
            addq.b #1,d0
            bra 12
2%:         rts
;
nlab2:      bra 1%              ;1% is undefined here
2%:         nop
nlab3:
```

During pass two a 'U' (Undeclared symbol) error will ensue if a local label does not exist within its defined scope.

3.5 Expressions

The assembler will accept any non-prioritized simple expression consisting of:

1. symbols
2. normal/local labels
3. denary/hexadecimal numbers
4. single character strings
   (Up-arrow facility, see Sec.3.6, is neither
   required nor permitted)

5. the operators:

+	Unary plus / Add
-	Unary minus / Subtract
*	Unsigned 16-bit Multiply
/	Unsigned 16-bit Divide
SHR	Shift right ('n' places)
SHL	Shift left ('n' places)
OR	Logical OR
AND	Logical AND
NOT	One's complement

NUMBERS

Numeric values may be defined either in denary or in hexadecimal. If hexadecimal is being used the number must be preceded by an ampersand (&) or a dollar sign ($):

For example:

```
defb 12,45,&3A
defw $E2,$3AB0
```

If the first digit following a $ or & hexadecimal delimiter is not a valid hexadecimal digit then an 'N' (Number format), or 'S' (Syntax), error will ensue.

SIMPLE EXPRESSIONS

A simple non-prioritized expression is defined in this case to mean any
expression of the general form:

<+/-> <operand> (<operator> <operand>)

A unary minus or plus may precede the first operand. Further operator-operand pairs may be used if desired. Expression evaluation is strictly from left to right. The NOT operator is a special case in that only one operand may exist, and this operand must be a symbol or a normal label. An 'I' (Illegal expression) error will ensue if the assembler cannot pass the expression in its context. In most cases this will also be followed by an 'S' (Syntax) error. Some valid examples are:

```
true   =  -1
false  =  not true
days   =  5
;
        prog:  moveq #true and &FF,d0
        moveq  #name and 255,d2
        moveq  #name shr 8,d3
        moveq  #'A',d0
        moveq  #'z'+1,d0
;
```

```
        moveq  #''',d0           ;Up-arrow (see 3.6)
        moveq  #'^',d0           ;equivalents, ie:
        moveq  #'A'+$80,d0       ;short form is used.
;
        moveq  #name/256+1,d2
        moveq  #days*24,d3
;
1%:     defb   0                 ;Data store
        move.w store,a0
;
store: defb 0,0
;
mask    = true shl 8 + 1
mask2   = mask or $2020
```

Expression values will take on an 8-bit, 16-bit, or 32-bit value depending upon the context of the expression. Assembler 'O' (Overflow) or ‘R! (Range) errors will ensue if it seems that an assignment is out of context (e.g., if a 16-bit value is being used in an 8-bit context). Some assemblers will simply assign the least significant bytes in such cases, which greatly increases the amount of debugging time required when you find out that your program does not work as you intended. For the purposes of conditional assembly, the expression will be deemed true if the most significant bit of the result is set (e.g., -1), or false if this bit is unset (e.g., 0).

3.6 Data definition

Data may be defined by using the following assembler pseudo-operators:

DEFB - Define byte / char (8-bit)
DEFW - Define word (16-bit)
DEFL - Define long-word (32-bit)

Alternatively data storage space may be allocated (as zeros) by using the pseudo-operator:

DEFS - Define space (n bytes)

The four data pseudo-operators available enable any form of static data storage to be defined, and may be used in the following ways.

DEFB

This pseudo-operator is used to define byte values and character

strings. A free integration of both types is permitted in any one definition line:

```
        defb 13,'This is a message',13,0
        defb 'ABCDEF'
        defb 0,1,2,3,4,5,6,7,8,9
```

Each element of the definition line is separated from the next by a comma (,). If the first character of an element is a single quote, a string of characters is assumed to exist up to, but not including, the next single quote ('). In the context of string definitions the

following is also applicable:

1. an up-arrow followed by a single quote will assemble as a single
   quote: defb '^''
2. an up-arrow followed by an up-arrow will assemble as a single
   up-arrow: defb '^^'
3. an up-arrow followed by any other character will force the most significant bit of that character to be set: defb '^A'

These special cases may exist anywhere with a string definition:

```
        defb 'A^BC'
        defb '^'up^''           ;'up' (with quotes)
        defb 'A^^2'
```

DEFW AND DEFL

These pseudo-operators force numeric definitions to occupy 16-bits (in the case of DEFW) or 32-bits (in the case of DEFL) whether or not the actual value could reside in an 8-bit location.

```
        defw 34,$56
        defl 900,$4B330,2
```

Strings (as defined under DEFB) may not be defined using these pseudo-operators. Each element in the definition line must be separated from the next by a comma (,).

DEFS

If an area of memory is to be allocated to some use, but the initial values within this area do not need to be specified (e.g., heap storage space), this pseudo-operator may be used. The single argument that must follow this operator will specify the number of bytes to reserve. The assembler fills the space with zeros.

3.7 Origin setting

The memory address where the assembled code is to start is defined by the ORG pseudo-operation:

```
        ORG $2A000
```

More than one ORG statement may exist within a program although it is illegal to define an origin which is lower in memory than the current assembly address. Previously declared labels or symbols may be used within an expression as an argument to ORG. For example, it would be possible to force an ensuing piece of code to reside at a clean page boundary:

```
        current:

                  ORG current+256 and $FFFFFFOO
        neode:
```

It is common practice, when writing executable code programs and extensions to SuperBASIC, to omit the ORG statement altogether. Assembly will then be based at address zero.

WARNING: Labels and symbols used in ORG expressions **must** be pre-defined. If this is not the case, different origins will exist during pass 1 and pass 2. In such cases the code will fail to assemble properly. The gap between the end of any previous code and a new ORG will be filled with zeros by the assembler.

3.8 Conditional assembly

Individual blocks of code may be conditionally assembled using the COND, ELSE, and ENDC pseudo-operators. The operator COND expects an expression as an argument. If the most significant bit of the result is set, the value is deemed true and the following code will be assembled.

Conditional assembly (or non-assembly) of code will continue up until the next ELSE or ENDC operator. If an ELSE operator is found, the condition for assembly is reversed, and the appropriate assembly continued up until the next ENDC operator. The particular level of conditional assembly is terminated on reaching the corresponding ENDC operator,

Conditional assembly may be nested. If pass 1 is completed, but nesting levels for conditional assembly have not been completely matched, a fatal 'Assembler error’ will ensue and assembly will cease (i.e., pass 2 will not be entered). A 'C' error will ensue if an ELSE or an ENDC operator is encounteted before a corresponding COND operator. Examples of this nesting are as follows:

```
              yes_please   =-l1
              no_thank_you = not yes please
        1.    cond yes please
                    subx d2,d0          ;assembled
              else
                    subx 4d0,d2         ;not—assembled
              endc
```

```
2.    addx dl1,d2 3            ;level 0
      cond no_thank_you
        addx d2,d3             ;level la
        cond true
          addx d3,d4           ;level 2a
        else
          subx d4,d3           ;level 2b
        endc
      else                     ;level 1b
        subx d3,d2
      endc
        nop                    ;Back to level 0
```

Note that the QRY form of defining symbol values as true or false (described in Sec.3.3), is an extremely useful mechanism for conditional assembly, for example, in cases where slightly different code needs to be generated depending on whether or not the code is to run in ROM. The actual source code need never be changed - it would simply be a matter of entering the appropriate responses at assembly time.

3.9 Directives

The assembler Supports a number of assembly directives, invoked by using an asterisk (*) as the first non-blank character in a statement line. The following are supported:

```
1. *Eject
2. *Heading <string>
3. *List    <on/off>
4. *Number  <on/off>
5. *Include <filespec>
```

All of these may be abbreviated to just their first character (for example, *E is the same as *EJECT).

***EJECT AND *HEADING**

*Eject causes a form-feed to occur in the list file, and the page number to be increased by one. Any heading, which had previously been defined, remains.

*Heading allows a heading message to be defined which will be used to document page headings in the list file. A form-feed will also occur automatically (as with *E). The maximum length of a heading is 35 characters, Headings longer than this will be truncated.

If one of these two directives is not given before a form-feed is due on a list file (in order to skip over pages in perforated listing paper), then the assembler will force a page throw as and when necessary (normally after 56 lines of assembly listing).

***LIST**

*List is used to turn the listing on and off. If the word ON follows the directive then the listing will be turned on. If the word OFF follows the directive then the listing will be turned off. Note that the directive *L ON will have no effect if the list-file device, specified in the original command line, was coded as null (Z). The directive is particularly useful for conditionally listing parts of a large source file. The symbol table is always produced if the list-file is active and therefore one way of getting just a symbol table as the list output is to (conditionally) set the list directive off at the beginning of the source:

```
FLST QRY Full listing required
;

      cond not FLST
*L off
      endc
;
<Symbol table produced anyway!>
```

***NUMBER**

*Number has the same syntax requirements as *List. The directive enables the generation and printing of line numbers within the list file to be switched on and off. The normal state is for line numbers to be given.

***INCLUDE**

*Include requires a full file specification as its argument. The specified file will be included in the source input stream at that point in the assembly. This feature enables a suite of library sources to be kept on a Microdrive cartridge and included in a program as and when required.

Only one level of inclusion is allowed and a file will fail to be included if its *I directive is within an already included file. In such cases an 'F! (File inclusion) error will ensue and assembly will continue at the next line in the current source file.

If a file cannot be opened because, for example, the file specification is incomplete or wrong, an error message will be given and assembly will stop. Note that the file specification must be the same as that which would be given to access a Microdrive under SuperBASIC. There are no restrictions on extensions, as is the case within command line specifications.

It is normal practice with large source,documents to have one (short) main module which *Includes all other external modules that are required.

3.10 Alternative mnemonics

A set of alternative mnemonics exists within the assembler to aid the programmer both in terms of style and readability. First is the mnemonic for '‘exclusive-or' operations. There are two widely used mnemonics for this instruction and both are supported:

Standard	**Alternative**
EOR	**XOR**

Second, there is the common confusion, especially with processors that cater for signed and unsigned arithmetic, as to the true interpretation of the 'carry-clear' and 'carry-set' conditional statements. As such the assembler provides the following:

Standard	**Alternative**
BCC, BCS	**BES, BLO**
DBCC, DBCS	**DBHS, DBLO**
SCC,SCS	**SHS, SLO**

The mnemonic part '‘HS' Stands for ‘higher or same', and 'LO' stands for ‘lower’, They differ from the 'greater or equal' (GE) and ‘less than' (LT) mnemonics in that they refer to conditions set after an unsigned operation.

3.11 Error messages

The assembler performs many checks while running and a number of errors and list-file error codes will occur if the source is illegal in some way. The error codes and messages which exist are as follows:

N> Number format error. A hexadecimal number is illegal.

L> Label format error. The format of a normal or local label is incorrect.

S> Syntax error. A catch-all message for lines which contain some form of illegal syntax.

M> Multiple definition. An attempt is being made to redefine a label or symbol during pass l.

I> Illegal expression. The arithmetic or logical expression is illegal within the context given.

U> Undeclared identifier. During pass 2 a symbol or label is being referenced which was not defined during pass l.

O> Overflow/Branch out of range error, A 16-bit value is being assigned to an 8-bit location, or a relative branch is out of range.

C> Conditional assembly error. An ELSE or ENDC operator was found before a corresponding COND.

F> File inclusion error. More than one level of file inclusion is being attempted.

R> Range error. An out-of-limits range is being specified within a particular instruction.

GENERAL ERROR MESSAGES

A few other errors may occur, usually fatal in effect. If a file cannot be opened or a Microdrive cartridge error occurs, an appropriate message is displayed and assembly will cease. If bad conditional assembly exists in pass 1, an error message is displayed and pass 2 is not entered. In all these fatal cases the error message will indicate the nature of the fault.

3.12 Word boundary alignment (ALIGN)

The 68000 processor will always require a word or long-word of data to begin on a word boundary (i.e., an even memory address). This implies that any instruction opcode must also be on a word boundary. When the assembler DEFB or DEFS pseudo-operators are used, the location counter could point to an odd address at the end of the definition line. If a 68000 instruction, DEFW line, or DEFL line immediately follows the definition, the resultant object code will not execute as expected. The 68000 will enter an error type exception process when an attempt is made to access any instruction or word of data at an odd address.

To stop you from having to count byte definitions, in order to make sure there are an even number of bytes defined (and getting it wrong!) the assembler pseudo-operator ALIGN is provided. This operator should follow any byte definition line that must, because of what follows, leave the location counter at an even address. For example:

```
          :
          datl:    defb 6,'FREEIT'
                   align
          dat2:    defw first,last,max,min
          :
```

If the location counter is incremented internally, to produce alignment, the byte skipped over will be set to zero by the assembler.

3.13 Aborting the assembler

If at any stage during an assembly operation you decide that you do not want to proceed to the end, you can abort the assembler by pressing the ESC key. The assembler will stop at the end of the current source instruction parse, clear up any channels which were open, and display the message:

```
Assembler aborted
```

The assembler should not be aborted by resetting the QL.

Appendices

Appendix A — 68000 INSTRUCTION SET SUMMARY

A.1 Addressing modes

Six basic addressing modes in the 68000 give rise to 14 actual modes. The modes of addressing are shown in Fig.A.1, together with the appropriate assembler syntax.

MODE	SYNTAX
Implied	
Register	SR, CCR, USP, PC
Immediate	
Immediate	#n
Quick immediate	#b
Absolute	
Short	a16
Long	a32
Register Direct	
Data register	Dn
Address register direct	An
Register Indirect	
Address register	(An)
Postincrement	(An)+
Predecrement	-(An)
Address register with offset	d16(An)
Register with index and offset	d8(An,i)
Program Counter Relative	
Address register with offset	d16(PC)
Register with index and offset	d8(PC,i)

Notes:

b = 3, 4, or 8 bits	i = An or Dn
n = 8,16, or 32 bits	An = address register
d8 = 8 bit offset	Dn = data register
d16 = 16 bit offset	PC = current location
d16 = 16 bit address	SR = status register
a32 = 32 bit address	CCR = condition codes
	USP = user stack ptr

Figure A.1 68000 addressing modes

A.2. Condition codes

There are three instructions (Bcc, DBcc, and Scc) which use a set of conditional tests. The tests are given ‘one/two character’ mnemonics and the full instruction mnemonic consists of the above names with 'cc' replaced by the test mnemonic (e.g., BHI, BF, DBEQ, SNE, and so on).

Each test produces a true or false result depending on the state of given condition flags in the 68000 CCR register.

In the table below, the alternative mnemonics are given in parenthesis after the standard mnemonic.

Mnemonic	Test	Interpretation
T	**1**	**true (always)**
F	**0**	**false (always)**
HI	**not(C).not(Z)**	**higher (unsigned)**
LS	**C+Z**	**less than or same (unsigned)**
CC (HS)	**not(C)**	**carry clear (unsigned)**
CS (LO)	**C**	**carry set (unsigned)**
NE	**not(Z)**	**not equal**
EQ	**Z**	**equal**
VC	**not(V)**	**overflow clear**
VS	**V**	**overflow set**
PL	**not(N)**	**plus**
MI	**N**	**minus**
GE	**not(N xor V)**	**greater than or equal (signed)**
LT	**N xor V**	**less than (signed)**
GT	**not(Z+(N xor V))**	**greater than**
LE	**Z+(N xor V)**	**less than or equal**

A.3 68000 instruction set summary

In Fig.A.2 (below) the instruction set of the 68000 MPU is given in alphabetic order. The effect of each instruction on the CCR flags is supplied, together with an indication of whether or not the instruction is privileged (i.e., can only be executed while the 68000 is in supervisor mode), and whether or not a data qualifier (i.e., .B, .W, .L, or .S) is normally used. Within the condition code list, the following key is used:

x : flag is affected
u : flag is undefined
- : flag is unaffected
0 : flag is reset to zero
1 : flag is set to one

The privileged instruction column (P) uses the following key:

n : not a privileged instruction
y : privileged instruction
? : privileged under certain conditions

If a '?' does appear in the 'P' column, reference should be made to an appropriate text book in order to determine which special cases can occur.

The data qualifier column (Q) uses the following key:

n : no qualifier used
y : qualifier used (or .W or ‘branch long’ assumed)
? : variable parameters depending upon use or non-use of data qualifiers

If a '?' does appear in the 'Q' column, reference should be made to an appropriate text book in order to determine which cases can occur.

		X N Z V C	P	Q
ABCD	Add decimal with extend	x u x u x	n	n
ADD	Add	x x x x x	n	y
	(When destination is 'An')	- - - - -	n	y
ADDQ	Add quick	x x x x x	n	y
ADDX	Add with extend	x x x x x	n	y
AND	Logical AND	- x x 0 0	?	y
ASL	Arithmetic shift left	x x x x x	n	?
ASR	Arithmetic shift right	x x x x x	n	?
Bcc	Branch conditionally	- - - - -	n	y
BCHG	Bit test and change	- - x - -	n	n
BCLR	Bit test and clear	- - x - -	n	n
BRA	Branch always	- - - - -	n	y
BSET	Bit test and set	- - x - -	n	n
BSR	Branch to subroutine	- - - - -	n	y
BTST	Bit test	- - x - -	n	n
CHK	Check regs against bounds	- x u u u	n	n
CLR	Clear operand	- 0 1 0 0	n	y
CMP	Compare	- x x x x	n	y
CMPM	Compare memory	- x x x x	n	y
Dbcc	Dec. and branch cond.	- - - - -	n	n
DBRA	Decrement and branch always	- - - - -	n	n
DIVS	Signed divide	- x x x 0	n	n
DIVU	Unsigned divide age	- x x x 0	n	n
EOR	Exclusive OR	- x x 0 0	?	y
EXG	Exchange registers	- - - - -	n	n
EXT	Sign extend	- x x 0 0	n	y
JMP	Jump	- - - - -	n	n
JSR	Jump to subroutine	- - - - -	n	n
LEA	Load effective address	- - - - -	n	n
LINK	Link stack	- - - - -	n	n
LSL	Logical shift left	x x x 0 x	n	?
LSR	Logical shift right	x x x 0 x	n	?
MOVE	Move	- x x 0 0	n	y
	(When dest. is 'An')	- - - - -	n	?
	(When dest. is 'CCR')	x x x x x	n	n
	(When src. is 'SR')	- - - - -	n	n
	(When dest. is 'SR')	x x x x x	y	n
	(When 'USP' used)	- - - - -	y	n

		X N Z V C	P	Q
MOVEM	Move multiple registers	- - - - -	n	y
MOVEP	Move peripheral data	- - - - -	n	y
MOVEQ	Move quick	- x x 0 0	n	n
MULS	Signed multiply	- x x 0 0	n	n
MULU	Unsigned multiply	- x x 0 0	n	n
NBCD	Negate decimal with extend	x u x u x	n	n
NEG	Negate	x x x x x	n	y
NEGX	Negate with extend	x x x x x	n	y
NOP	No operation	- - - - -	n	n
NOT	One's complement	- x x 0 0	n	y
OR	Logical OR	- x x 0 0	?	y
PEA	Push effective address	- - - - -	n	n
RESET	Reset external devices	- - - - -	y	n
ROL	Rotate left	- x x 0 x	n	?
ROR	Rotate right	- x x 0 x	n	?
ROXL	Rotate left through extend	x x x 0 x	n	?
ROXR	Rotate right through extend	x x x 0 x	n	?
RTE	Return from exception	x x x x x	y	n
RTR	Return and restore	x x x x x	n	n
RTS	Return from subroutine	- - - - -	n	n
SBCD	Subtract decimal with extend)	x u x u x	n	n
Scc	Set conditional	- - - - -	n	n
STOP	Stop	x x x x x	y	n
SUB	Subtract	x x x x x	n	y
	(When destination is 'An')	- - - - -	n	y
SUBQ	Subtract quick	x x x x x	n	y
SUBX	Subtract with extend	x x x x x	n	y
SWAP	Swap data register halves	- x x 0 0	n	n
TAS	Test and set bit 7	- x x 0 0	n	n
TRAP	Trap	- - - - -	n	n
TRAPV	Trap on overflow	- - - - -	n	n
TST	Test	- x x 0 0	n	y
UNLK	Unlink	- - - - -	n	n

Figure A.2 68000 instruction set summary

Appendix B — EDITOR/ASSEMBLER QUICK REFERENCE GUIDE

The editor and assembler packages are discussed in Chapters 2 and 3.

Given here are quick reference guides for their use.

EDITOR REFERENCE GUIDE

a) Top-level commands:

```
^E - Execute extended command:
   - D - Delete block (marker to cursor inclusive)
   - F - Find text string
   - I - Include external file (at cursor)
   - M - Move to line absolute
^F - Finish - return to SuperBASIC
^H - Give help on cursor control and deletion commands
^L - Load a file in from Microdrive
^M - Set current cursor line as marker line
^S - Save editor buffer on to Microdrive
```

b) Cursor control commands:

KEY	**NORMAL**	**SHIFT**	**ALTMODE**
up	line	page	start of text
down	line	page	end of text
left	character	word	start of line
right	character	word	end of line

c) Text deletion commands:

KEY	**CTRL**
up/down	delete current line
left	delete char/gap left
right	delete cursor char/gap

ASSEMBLER REFERENCE GUIDE

a) Comments:

Must be preceded by a semi-colon (;)

b) Labels:

Must be followed by a colon (:)

c) Directives:

i)	*EJECT	- Force a new page
ii)	*HEADING	- Create new heading and new page
iii)	*LIST <on/off>	- Switch listing file on/off
iv)	*NUMBER <on/off>	- Switch line numbers on/off
v)	*INCLUDE <file>	- Include external source file

d) Pseudo-operators:

i)	EQU (=)	- Static equate
ii)	QRY	- Dynamic equate
iii)	ORG	- Set program counter
iv)	ALIGN	- Align to word boundary
v)	COND <expr>	- Conditional assembly
	ELSE	
	ENDC	

e) Expression operators:

+	add	SHR	shift right
-	subtract	SHL	shift left
*	multiply	OR	logical ''or'
/	divide	AND	logical 'and'
		NOT	one's complement