Published in the United Kingdom by:
Adder Publishing Limited,
P.O.Box 148, Cambridge CB1 2EQ

Second edition January 1985

Because of reliability problems with microdrive cartridges, we have made it very easy to produce backup copies of our assembler. You are encouraged to make a backup copy before starting to use the assembler. However, we would like to remind you that it is **illegal** to copy programs for use by friends.

Contents

1 Introduction

The ADDER editor/assembler is a complete 68000 development package for the Sinclair QL computer. It allows assembler programs to be edited, assembled and run in a convenient manner. The entire system of editor, assembler and debugger all resides in memory together, so there is no slow accessing of microdrives to worry about. Rapid development of small assembler programs is therefore very quick and easy, and much larger programs can be supported by the ability to include external library files and macros. This assembler provides a full specification Motorola compatible MACRO assembler.

Programming in 68000 assembler allows many exciting possibilities to be explored. However, because the 68008 is such a powerful microprocessor, it is impossible to provide full technical information about programming it in a manual like this. Two other books should therefore be considered as essential reading and reference material. The **MC68000 16/32 Bit Microprocesor Programmer's Reference Manual (4th edition, ISBN 1 356 6795X)** published by Prentice-Hall covers the 68008 microprocessor in depth. It is also necessary to understand the operating system environment on the QL computer. All the required information about QDOS can be found in **QL Advanced User Guide by Adrian Dickens (1st edition, ISBN 0 947929 00 2)** published by Adder Publishing Limited, Cambridge.

1.1 General overview

The entire development package resides in memory together, but the functional aspects can be split into three distinct categories:

The Editor

The assembler source program needs to be entered into the QL before the assembler can be invoked. Programs are entered as normal text using the integral full screen editor. It can be used as a quick general purpose text editor in its own right if required. Either of the QL's screen modes (4 colour or 8 colour) are supported, as are the TV and monitor style displays. The following commands are provided:

normal characters	- characters you type are inserted in the text
left/right/up/down	- cursor keys move cursor by one space/line
SHIFT left/right	- move to beginning or end of current line
SHIFT up/down	- move to previous/next page
ENTER	- split current line or start a new line
TAB	- move to next TAB column
CTRL left/right	- delete left/right character or join lines
CTRL up/down	- delete/undelete lines (allows blocks of text to be moved around within the file).
CTRL SHIFT down	- COPY line of text into deletion buffer
F1	- redraws the screen
F2	- move to next error and redisplay error message (makes finding errors effortless)
F3 linenumber	- move directly to a numbered line (entered by itself moves to first line)
F4 filename	- insert named file in text at cursor position
F5 string	- search for string in source text
SHIFT F5	- find next occurance of the string
ESC	- exit back to command mode
SHIFT ESC	- exit editor without emptying deletion buffer

The Assembler

Once the source text for an assembler program has been entered into the QL, it must be assembled into an object code file. The assembler is compatible with Motorola's standard, although there are some deviations. The assembler makes a single pass over the source text so as to reduce the assembly delays. In spite of this, all problems concerned with forward references are resolved correctly. Included within the assembler is the facility for full symbol arithmetic using relative and absolute 32 bit symbols. All 68008 instructions and addressing modes can be assembled, and error messages are displayed as they are located by the assembler. The position of errors are recorded so that these can easily be located at a later time from within the editor. The assembler fully supports MACROs and CONDITIONAL ASSEMBLY.

Command mode

In this mode, the user can select the editor, assemble and run programs and perform various useful debugging tasks. The main commands are:

E - enter the editor
A - assemble the current text
R - run the assembled program

Other commands allowing various auxiliary operations:

8 - switch to 8 colour QL screen mode (large characters)
4 - switch to 4 colour QL screen mode (small characters)
S - save the text in the current filename
N - new name for current text
F - switch to editing a new file
P - print a value or expression, including assembler symbols, on the screen or printer (or any other channel)
D - dump a portion of the compiled program (or any other part of memory) onto screen or printer. Dumping one of the registers causes all registers to be dumped for debugging purposes.
T - type the current text
L - produce a listing with line numbers and hex code
O - change output file for P, D or T operations
B - set the baud rate for the serial line
Q - quit the assembler system and return to BASIC
U - update (POKE) memory locations or modify registers
M - LOCK or UNLOCK symbol table (use with 'patch' directive)

The whole system is presented in a number of screen windows which make it very clear which mode the system is operating in.

1.2 Loading the editor/assembler from microdrive

The remainder of this User Guide acts both as a reference and tutorial guide. In order to obtain the greatest benefit from the information which follows, you are advised to load up the assembler and try out the various features as they are discussed. You are also strongly advised to **MAKE AT LEAST ONE BACKUP COPY** of the assembler before proceding. Backup copies should be made as follows:

1. Place a blank cartridge in drive 2

2. FORMAT MDV2_ASSEMBLER

3. Place master cartridge in drive 1

4. COPY MDV1_ASM TO MDV2_ASM

5. COPY MDV1_ASMTEXT TO MDV2_ASMTEXT

6. COPY MDV1_ASMHELP TO MDV2_ASMHELP

Sometimes, the master may fail to load if the microdrive read heads are out of alignment. If the master fails to copy in this way, try the same procedure as before, but copying the master from mdv2_ to mdv1_. Now put the original master in a safe place where it will not be used, and work from the copied cartridge. First of all, the assembler needs to be initialised. To do this, type:

```
exec_w mdv1_asm
```

The assembler/editor will load into memory, then a large ADDER LOGO will appear on the screen. Wait for a few seconds whilst the entire system initialises itself, then you are ready to start programming. Note that the assembler could also have been entered using exec mdv1_asm. This would cause BASIC and the assembler to exist concurrently, CTRL-C being used to toggle between BASIC and the assembler.

The display which you should now see will consist of some five different windows. Initially, the system starts off in the command mode, so a flashing cursor should be visible in the command window (across the base of the display). The exact layout of the display depends on the mode (TV, monitor, 4 colour or 8 colour). The five windows are:

1. **Command window** - for entering commands

2. **Text window** - takes up most of the screen - this is where the text editor operates

3. **Line number display** - always displays the line number which the editor is at, or the number of the line which has been reached by the assembler during an assembly.

4. **Filename display** - always indicats the current filename

5. **Memory free** - indicates number of spare bytes of memory. On a standard QL, this starts at about 40000 bytes. This space is used for the program text, assembler's symbol table and the assembled program.

The filename display will show 'mdv1_asmtext'. This file is always loaded in at initialisation. If there is no such file on the microdrive, an empty file with this name will be set up.

The most important functions of the whole package are to edit text and to assemble 68000 programs. Other minor tasks can also be carried out, such as loading or saving files, producing assembler listings on a printer etc. These minor tasks are all carried out in command mode.

2 COMMAND MODE

This section explains all of the commands available in command mode. When the cursor is visible in front of the >>> prompt, this means that the system is in command mode. Commands are given by pressing a single key. Since there are quite a lot of commands, it is sometimes difficult to remember them all. A full list can be obtained by pressing either 'H' or '?'. The 'HELP' command simply prints out the contents of file 'mdvl_asmhelp'. This is supplied with a list of the commands, but can easily be edited to suit an individual's own preferences.

A - Assemble the current text

This runs the assembler on the current text object (more about this later) and produces:

a symbol table
a list of error messages for the text editor
a code object

As the assembler runs, the current line number, space remaining and the source file currently being scanned are constantly displayed at the top of the screen. If you wish to abort an assembly, press ESC and hold it down until the assembler grinds to a halt with the message **BREAK.

During assembly, error messages may be produced, and these are printed on the main text window. Do not worry about making a note of the offending lines because the editor remembers them. This makes it very easy to enter the editor after assembly and move to each of the errors in turn simply by pressing F2. If the assembler encounters more than 20 errors, it will automatically abort the assembly to allow the programmer to correct them.

The assembler is supplied with a very simple one line program in the default file 'mdvl_asmtext'. This should be displayed on the screen at initialisation, and contains the following text:

```
*
*       Example to multiply register d1 by d2
*       leaves the result in d2
*
        MULU    d1,d2
```

You can see how the assembler operates simply by typing A. The prompt 'Assembling mdvl_asmtext' appears in the main text window, and the line count rapidly increments until 'No errors' appears to indicate that assembly of the file was completed without any errors.

Syntax: Assemble

B - set the serial baudrate

This command sets the baud rate for the serial lines. It is normally used to set up the printer for output using the O command. The baudrate is selected using a single numeric digit (the first digit of the standard baud rate required). Hence the following should be used:

type	for
3	300 baud
6	600 baud
1	1200 baud
2	2400 baud
4	4800 baud
9	9600 baud
0	19200 baud

Syntax Baudrate: n

Example Baudrate: 3

D - dump an area of memory or registers

This command has two different modes of operation:

MEMORY DUMP

Type D followed by address,n where n is the number of **words** to dump. Both parameters should be expressions in a form acceptable to the assembler. The address may be relative or absolute. The corresponding area of the QL's memory is dumped to the screen or printer in hex and character forms. This feature allows the user to examine the hex of an assembled program, or other areas of the QL's memory (see also the **L** command). Pressing ESC during a dump will interrrupt the dump operation and return the system to command mode.

The hex of an assembled program can easily be typed out by putting a label at the beginning and end of the program. If these labels were **startlab** and **endlab** respectively then the program could be dumped as startlab,(endlab-startlab)/2. The division by 2 is required to give the parameters in words rather than bytes. Note that the start address must therefore be even so that a dump always starts from an even word boundary. Alternatively, it is possible to print the value of *, which points to the end of the program, and dump startlab,(*-startlab)/2.

The width of the dump is determined by the width of the screen, regardless of whether the output is being sent to the screen, printer or a microdrive file. Putting the screen into 4 colour mode causes more bytes to be dumped per line than 8 colour mode. Some experimentation is probably worthwhile to find the optimum width for your printer.

Syntax Dump: address,number of words

Example Dump: $C44,15

Note that addresses are assumed to be in decimal unless preceded by a '$', in which case they are in hexadecimal.

REGISTERS DUMP

If a register is given in place of the address parameter, all of the 68008 registers (excluding the stack register A7) will be displayed on the screen. This facility is extremely useful for trying out short programs. The update **U** command can be used to modify the values of the registers before a piece of code is executed, then the dump command can be used to examine the register contents on return.

Syntax: Dump: register

Example: Dump: d0

E - enter the EDITOR

This causes the editor to take over control, allowing text files to be typed in and modified. Editing programs in the editor is explained in more detail later on. You can get back into command mode from the editor by pressing the ESC key.

Syntax: Edit

F - enter new Filename for current text file

This command should be used with care because it DISCARDS THE CURRENT TEXT and loads in the named file. Loading of new text will occur as soon as the source file name has been entered. If you press the **F** command key by mistake, simply type ENTER and no new text will be loaded and the old text will still be there.

Syntax: Filename: new filename

Example: Filename: mdv1_scratch

H - HELP

The **H** command is synonomous with the **?** command, and will cause the file 'mdv1_asmhelp' to be loaded from microdrive and displayed on the screen. This file usually contains a summary list of the commands which are available. However, individual users may find it helpful to put different comments in this file, so it can be edited in the normal way using the editor.

Syntax: H (gives message 'Getting Help..')

L - Generate assembly Listing of program

The **L** command allows assembled programs to be printed on a printer, complete with line numbers and hex codes.

Because this assembler is a single pass type of assembler, the **L** command requires both a text source file and and object code file. If the object code file does not yet exist on the microdrive, the required text file should be assembled and then saved using **W** command. The **L** command can then be invoked. You will be asked to provide the 'Binary input file:' which has just been saved on the microdrive.

A complete listing of the assembled program is then produced. Output will appear on the screen as default, but can be redirected to the printer using the **O** command.

Note that each MACRO definition will be expanded, and that line numbers within a MACRO will always start from 1. The LIST and NOLIST directives can be used effectively in such cases. Whenever the NOLIST (or NOL) directive is encountered, the listing is suppressed. The assembler text following the NOLIST is therefore not printed out. Printing out of text can be recommenced by inserting a LIST directive at the relevant point within the text.

Syntax: L (gives message 'Assembly Listing.')
Binary input file: object file name

Example:
```
A
W mdv1_testfile_obj
L
Assembly Listing.
Binary input file: mdv1_testfile_obj
```

M - LOCK or UNLOCK the symbol table

Whenever a file is assembled, the assembler generates a 'symbol table'. This table contains the type and value of all the symbols, like labels and variables, which were used in the assembled file. Normally, the symbol table is deleted whenever the file is assembled. This ensures that the assembler will not be confused by the old values which were assigned on the previous assembly.

When very long files are being assembled, the assembly time can be appreciable, especially if most of the source text is stored on a microdrive cartridge. This is where the 'patch' directive becomes useful. Small modifications, like the condition for a branch instruction can easily be modified **without** re-assembling the entire file (see 4.9).

The 'patch' directive requires that the original symbol table should remain intact between assemblies. The symbol table can therefore be LOCKed using the **M** command. Once locked, the original symbol table cannot be deleted until it is unlocked again using the **M** command.

Syntax: M

if unlocked: L to Lock syMbol table:
if locked: U to Unlock syMbol table:

N - New Name for the current text

This changes the name under which the current text file will be saved using the S command. The current text is not changed in any way. Using this method it is possible to duplicate files on microdrives by loading one in, changing its name and saving it again. Alternatively, the new version number for a particular file can be substituted for the old one. This will then ensure that a different file is saved, so in the event of a microdrive cartridge failure, there will still be an uncorrupted version available.

Syntax: New name: file spec.

Example: New name: mdv1_test5_asm

O - select new Output stream

Output from the **D, P, L** and **T** commands normally goes to the screen text window. Using this command, the output can be redirected to a microdrive file or device such as a printer. After an **O** has been typed in command mode, the system prompts for a new output filename. The possible responses are:

just press ENTER - has no effect (keeps original output)
type '*' then ENTER - returns output to default text screen

or alternatively attempt to open a new output channel to which all future output will be directed.

Syntax: O (gives prompt 'spool Output to:')
spool Output to: file spec.

Examples:
```
O
spool Output to:ser2          (serial port)
or
spool Output to:*             (screen console)
```

Note that the baud rate for the serial port can be changed using **B.**

P - Print a value

This is like a sophisticated desk calculator. Type a value or expression: any value can be specified, relative or absolute, making full use of the assembler symbol table and expression facilities. The value will be printed on the screen in hex, decimal (and if appropriate ASCII character representation). A list of values can be specified, separated by commas (but no spaces).

Syntax: Print: numeric expression

Examples:
```
Print: 65
Print: $EE56,78
Print: var1
```

Q - Quit the assembler system

This command returns control to BASIC. In case you press the Q key by mistake, you are asked 'sure?(Y/N)'. An 'N' in response to this question simply cancels the Q command and returns to the assembler system. An 'Y' in response to the question closes down the assembler, enters BASIC and

returns all memory used by the assembler to QDOS. The only way to re-enter the assembler is to reboot from microdrive. If the option to switch between BASIC and the assembler is required, the assembler should be initialised with EXEC mdv1_asm. CTRL-C then toggles between BASIC and the assembler.

Syntax: Quitting - sure?(Y/N): Y or N

R - Run the code object

This runs the assembled program. It is wise to save the original source text to microdrive before running a program to ensure that any recent edits to the source text are safe. If there are any errors in the assembled code, the entire machine could possibly crash.

The assembler automatically inserts an RTS instruction at the end of an assembled object file. The single line example program to multiply two numbers together (see under **A** command) can be run in this way. Note that the values to multiply should first be set using the **U** command.

Certain types of assembled programs are unsuitable for running directly. Those which are designed as additional BASIC functions and procedures for example will have to be saved on microdrive initially, then loaded later under BASIC. Programs which are completely self-contained are often suitable for multi-tasking. Such programs need to be EXECed from BASIC and can be stored using the EXEC option of the **W** command.

The register values which are given to the assembler program at 'run time' can be set using the **U** command. Before returning to the assembler, the returned values of the registers are saved so they can be examined using the **D** command.

Syntax: Run

S - Save the current text object to microdrive

This causes the current text to be saved to microdrive. The filename used is displayed on the screen under the heading 'FILE:'. This name can be changed using the **N** command. Use the **S** command regularly to ensure that there is always a current version of the program on microdrive, otherwise a crash may cause all the current text to be lost.

Syntax: S

responds with 'Text saved as mdv1_asmtext'

T - Type out the current program

This causes the current program to be listed to the currently selected output channel. It is normally only used with the printer selected as output because it is generally much easier to look at files using the editor. ESC can be used to terminate the typing of a program.

Note that the **L** command also produces listings, but these listings contain line numbers and hex codes. Typed programs are just the straight text file as seen by the editor.

Syntax: Type text

U - Update the QL's memory or registers

This command has two functions. The first function allows memory to be changed directly, in a similar way to the BASIC POKE command. The second function allows the values in the registers when the assembled program is run to be modified.

After U has been pressed, the computer will ask you if you want to write a byte, word or long word. Type B, W or L as appropriate.

The computer then responds with the query 'ad,values: '.

Memory modification

For the first type of POKE operation, type in the first address to be modified followed by the value at that address and as many subsequent addresses as are required, all separated by commas.

Register modification

To change a register, simply type the standard register code, like 'd2' followed by data separated by commas. If more than one data parameter is provided, it will be put into the next register in order. The assembler remembers these values for the registers until the assembled code is run. These values are then passed to the assembled program just before it is entered. The sequential order for entry is:

d0,d1,d2,d3,d4,d5,d6,d7,a0,a1,a2,a3,a4,a5,a6

The demonstration multiplication program can now be tried out. Type:

Update Byte/Word/Long: L
ad,values: d1,2,3

This will load d1 with 2 and d2 with 3. Dump the registers using the **D** command to show this. Now run the program using **R** command. Then dump the registers again using D. You will notice that the register d2 now contains the value '6'. The program has multiplied two numbers together. Typing **R** again will cause d2 to be multiplied by d1 again. Looking at the registers will then reveal that d2 contains the hexadecimal value $C which is equivalent to decimal '12' (a $ before an argument indicates that the number is in hexadecimal, otherwise decimal is assumed).

The ability to set up all the registers, try out a small piece of code and then look at the results is a valuable tool for use in debugging programs.

All assembler arithmetic and symbol table facilities can be used with this command. The overall result is very similar to the DC directive in the assembler, except that the values go to a position in memory that starts at the stated address rather than in the program space.

Syntax: Update Byte/Word/Long: size specifier
ad,values: address,value1,value2,....

Example: Update Byte/Word/Long: W
ad,values: $20000,$AA,34,56 (loads into screen memory)
or ad,values: d1,4,5 (puts 4 in d1 and 5 in d2)

W - Write the assembled program to a named file

This command writes out the assembled code to a named file. The command will prompt you for a filename and then write to that file.

Machine code routines which are designed as additions to BASIC, or as executable programs in their own right as Jobs can be saved with this command. Multi-tasking programs can be saved using the W command by specifying the data space required.

Syntax: W filename (saves object code in a file)
or W filename,dataspace (saves as EXECable file)

Example: W responds with
Save in file: mdv1_code
or Save in file: mdv1_mon_exec,100

> - Print status of symbol table and version number

This command prints out information about the assembler version and informs the user whether the symbol table is LOCKED or UNLOCKED.

Example: > responds with
ADDER QL Assembler
Version: 20th November 1984
no symbol table

? - HELP

This command operates identically to the H command.

Syntax: ?

4 - Reinitialise the screen in 4 colour mode

This resets the assembler to use small characters. This is highly preferable for serious work because it provides more characters across the screen than the equivalent 8 colour mode. The exact number of charcters displayed across the screen in mode 4 depends on whether the monitor or TV type of display was selected at power up.

Some displays are capable of displaying the smaller characters clearly, but some characters are lost off the edge of the screen. If this is the case, you should select the TV option at power up.

Syntax: 4

8 - Reinitialise the screen in 8 colour mode

This screen mode uses larger characters than mode 4, which may appear much clearer on some televisions. Unless it is difficult to read the mode 4 characters, it is unusual to use mode 8 for any serious work.

Syntax: 8

3 EDITING FILES

3.1 General introduction to the text editor

The assembler source program text needs to be entered into the QL before the assembler can be used. Programs are entered as normal text using the integral full screen editor. This editor can be used as a quick general purpose text editor in its own right if required. Either of the QL's screen modes (4 colour or 8 colour) are supported, as are the TV and monitor style displays. The following commands are provided:

normal characters	- characters you type are inserted in the text
left/right/up/down	- cursor keys move cursor by one space/line
SHIFT left/right	- move to beginning or end of current line
SHIFT up/down	- move to previous/next page
ENTER	- split current line or start a new line
TAB	- move to next TAB column
CTRL left/right	- delete left/right character or join lines
CTRL up/down	- delete/undelete lines (allows blocks of text to be moved around within the file)
CTRL SHIFT down	- COPY line into deletion buffer
F1	- redraws the screen
F2	- move to next error and redisplay error message (makes finding errors effortless)
F3 linenumber	- move directly to a numbered line (F3 enter moves to top of source file)
F4 filename	- insert named file in text at cursor position
F5 string	- search for string in source text
SHIFT F5	- search for next occurance of string
ESC	- exit back to command mode
SHIFT ESC	- exit editor without emptying deletion buffer

3.2 Available editing modes

Before you start editing, the optimum screen layout for your system should be selected. This will be determined by a number of factors. First of all, the monitor or TV type of display should be selected at power up. Take care with the monitor display. Some monitors cannot display the full width of the QL's screen, and have to be operated in the TV display mode. Now decide whether you require as much text on the screen as possible (mode 4) or the largest characters possible (mode 8). Mode 4 should normally be used unless you are using a television, in which case mode 8 may be more suitable. Select the correct mode by typing either '4' or '8' from command mode (see previous section).

3.3 Simple editing with examples

The commands available on the editor were summarised at the start of this section. We are now going to try out some of these features.

First of all, clear the text memory by creating a new edit file, called 'mdv1_temp' for example. This is created using the **F** command in command mode. The text window should now be cleared of any text.

Type **E** to enter the editor. The cursor (not flashing) will appear in the top lefthand corner of the screen. Try typing some text to get a feel for the editor's operation.

Characters typed at the keyboard will be inserted directly into the text at the current cursor position. The **ENTER** key inserts a newline character at the cursor position.

The arrow keys move the cursor left, right, up and down. The cursor will not move to any position where there is no text. Try moving the cursor to the middle of one of the lines of text which you have just typed. Now hit the **ENTER** key. The line is split at that point due to the insertion of a carriage return character. The lines can be joined again by CTRL-backarrow.

SHIFT-arrow keys can be used to move the cursor up a page, down a page or to the start or end of the current line.

The TAB key moves to the start of the next column of characters (each column is 8 characters wide). This is extremely useful for entering programs, because each type of entry can be put in a separate column, leading to neat program layout.

CTRL-left and CTRL-right delete a character forwards or backwards as is normal on the QL. CTRL-up, CTRL-down and CTRL-SHIFT-down can be used to move or copy blocks of text. The operation of these facilities is covered in the next section.

Lines can be of any arbitrary length, but the latter parts of the lines are not displayed if they protrude beyond the right hand edge of the screen. Characters there can be seen by splitting a line with the ENTER key.

The function keys are also used in the editor.

F1 redraws the text

F2 moves the cursor to the suspected location of the next error from the previous run of the assembler, and prints the relevant error messages on the screen. This is covered in a later section entitled 'Error correction after assembly'.

F3 number moves the cursor to the specified line. This can be used to move rapidly to anywhere in the file. For example, **F3 1 ENTER** will move to the start of the file and **F3 1000 ENTER** will move to the end of the file and so on. This function really becomes useful when editing fairly long files.

F4 filename inserts the contents of a named file into the text. This can be used to concatenate text files, or to copy blocks of text.

F5 string searches through the source text from the current position of the cursor until the designated 'string' is found.

SHIFT-F5 searches through the source text for the next occurance of 'string'. This is used in conjunction with the F5 search facility.

CTRL-F5 can be used in the normal way to temporarily halt output during listings, so that the text can be read.

The editor is exited by **ESC**. This causes the deletion buffer to be emptied and returns control to command mode. Return to command mode can also be effected using **SHIFT-ESC**, but the deletion buffer remains intact. Text stored in the deletion buffer can then be used when the editor is entered again.

There is no 'cancel edits' command - exiting the editor causes the updates to occur. This is because everything is being done in store, so there is not usually enough space for both old and new texts to coexist. The only way of cancelling edits is to back up to the last disc version (using the **F** command in command mode).

3.4 Block MOVE and COPY commands

Moving blocks of text is performed by 'deleting' lines into a deletion buffer, moving the cursor to the new destination and then 'undeleting' the lines at this new position. Large blocks of text can easily be moved around in this way.

To 'delete' a line into the deletion buffer, position the cursor on that line and type **CTRL-downarrow**. The line will disappear. Now move the cursor and type **CTRL-uparrow** and the line of text will pop up from the deletion buffer onto the screen. Any number of lines can be deleted and undeleted, providing a powerful block move facility.

Text can be copied in a similar manner. Position the cursor on the line to be copied and type **CTRL-SHIFT-downarrow**. The line will be entered into the deletion buffer, but the original will also remain. The copied line can then be undeleted anywhere in the text with **CTRL-uparrow**.

3.5 Error correction after assembly

The use of this facility will become clearer after you have read the section on assembling files. When a program is assembled, any syntactic or other errors are printed on the screen as the assembler finds them. After the assembler has located all of the errors, it is up to the user to edit and correct these errors.

Normally, this would require a note to be made of the offending line numbers and the type of error present. However, the Adder assembler records all occurances of errors during the assembly for you. In the editor, it is then simply a matter of pressing function key **F2** to search for the next error.

Pressing **F2** causes the cursor to be moved to the position of the next error, and also causes the error message for that error to be printed out on the screen. The programmer will then have to assess the cause of the error, correct it, and go on to the next error. This fast search facility allows programs to be debugged very rapidly.

4 ASSEMBLING FILES

4.1 General introduction to the Assembler

The ADDER assembler is a 68000 MACRO Assembler compatible with the Motorola standard. Its basic function is to convert an assembly language program, written in normal text into machine code suitable for direct execution by the 68008 microprocessor inside the QL.

The assembler is started by the **A** command from command mode. It then performs a single pass over the input text, generating code as it proceedes. Forward references are remembered and filled in as they become known. Unless otherwise specified, the assembler behaves according to the Motorola standard for 68000 assemblers. It was based on the information contained in the 3rd edition of the Motorola 68000 User's Manual.

4.2 Syntax

The general format of each line of input text must conform to:

EITHER
optional label
optional colon following label
one or more spaces
the opcode, possibly with size modifier
one or more spaces
the operands, with **NO SPACES!**
one or more spaces
comments.

OR
asterisk in column 1
comments.

OR
blank line (like comment has no effect on assembly)

Labels may be of any length. They should start with a letter, and continue using letters, dots, underscores and digits. The case of letters is not significant.

Opcodes consist of the legal 68000 instruction opcodes, or the directives listed in section 4.6. Case is not significant. Some opcodes may include size modifiers, which are of the form:

.L	32-bit (Long word) modifier
.W	16-bit (Word) modofer (this is the default size)
.B	8-bit (Byte) modifier
.S	8-bit offset in branch instruction. If this is not specified, branch instructions generate 16bit offsets. .W may be quoted explicitly on branch instructions, to emphasise the 16-bit offset.

Some opcodes have several different forms. For example, ADD also has the versions ADDA, ADDI and ADDQ. As a general rule, the ...A and ...I forms of such instructions need not be specified because the assembler will automatically choose the correct form for the instruction. However, it is necessary to explicitly state the ...Q form since the assembler will not otherwise produce this form.

Example lines:

```
FRED    MOVEQ   #3,d1
*
*       This is a comment
        RTS     this comment follows an instruction

 MULS d1,d2 columns don't have to line up, but it looks neater

        clr.l   d1      lower case is equally valid
```

4.3 Dynamic Debugging tools

The Adder assembler provides several useful tools to aid in program development.

The first of these is the automatic error search facility within the editor. After a program has been assembled, producing a list of errors, the programmer can simply enter the editor and press function key **F2**. This moves the cursor to the first error and prints the relevant error message on the screen. Subsequent errors can be located in a similar manner by continually pressing **F2**.

Another powerful debugging aid is provided by the **U** (update registers or memory) and **D** (dump out register or memory contents) commands. These two commands allow the memory and registers to be configured in a special way, to try out specific bits of code. Memory contents and the values of all the registers can be preset to any desired values using the **U** command (see Command Mode section).

Upon return from running a particular routine, the registers and/or memory contents can be examined to ensure that the routine had the desired effect. An example of this was provided in the command mode section.

Other useful facilities include the **P** (print value) command, which allows any expressions from the symbol table to be printed out, **T** (type) and **L** (list with assembled codes) which allow the entire program to be printed out for examination.

During the edit/assemble/run cycle, it is often inconvenient to re-assemble the entire source file just to check a minor alteration. The 'patch' directive allows minor alterations to be tried out without re-assembly (see section 4.9)

4.4 Addressing modes

The operands which are provided in the source text are separated by commas (see assembler syntax), and are either expressions or addressing modes. The following forms of addressing modes are allowed:

form	example	addressing mode name
Dn	D1	data register direct
An	A3	address register direct
(An)	(SP)	address register indirect
(An)+	(A1)+	address register indirect with postincrement
-(An)	-(A4)	address register indirect with predecrement
a(An)	24*2(A1)	address register indirect with displacement
a(An,Xn)	12(A3,D1.L)	address register indirect with index
(An,Xn)	(A3,A2)	address register indirect with index (zero offset assumed)
a	lab3	absolute address (long or short)
r	relval	program counter relative with displacement
r(PC)	relval(PC)	program counter relative with displacement
r(Xn)	relv(D2)	program counter relative with index
r(PC,Xn)	relv(PC,D2)	program counter relative with index
#a	#23	immediate data

In the forms described in the above table, the following conventions are used:

An	means any address register, A0-A7 or SP (same as A7)
Dn	means any data register
Xn	means any register, with a possible .L or .W size modifier
a	means any absolute value of expression
r	means any relative value or expression

With PC index mode using an address register, the index size must be specified. This is because

```
MOVE.L  fred(A3),D1
```

is mistakenly read as an address register with displacement, even if fred is a relative value. If fred is a relative value and PCindex was intended, say:

```
MOVE.L  fred(A3.W),D1
```

or

```
MOVE.L  fred(PC,A3),D1
```

The syntax for long and short absolute modes is ambiguous. If there are no forward references then the smallest one that fits will be used. If there are forward references then only short absolute can be used, otherwise the value turns out to be too big and an error results.

4.5 Expressions

These may consist of symbolic names, operators, numeric constants and the * symbol. The following forms of constant are allowed:

Decimal

A sequence of decimal digits such as 1245

Hexadecimal

A dollar sign, followed by a sequence of hex digits such as $FF3

Character

Up to four characters enclosed in single quotes. A '' in the string counts as a single quote mark. Example: '3'

The character constant generates the corresponding ASCII code. All arithmetic is to 32bit accuracy, thus the limit to 4 charcters.

Symbolic names are values set using the EQU or SET directives and program labels. Forward references to values not yet defined may be used freely.

The * symbol generates a relative value, equal to the current position in the code.

All values are either relative or absolute. A relative value is an offset into the (relocatable) code section. There are limits on the ways in which relative and absolute values may be combined, as (for instance) multiplication on a relative value is unlikely to be useful.

```
relative + absolute  -> relative value
absolute + relative  -> relative value
relative - absolute  -> relative value
relative - relative  -> absolute value
```

These are the only ways in which relative values may be combined - all other arithmetic is constrained to be between absolute values.

The following operators are available:

+	2s complement addition
-	subtraction or negation
*	signed multiplication
/	signed division
<<	shift left - a<<b is 'a' shifted to the left by 'b' places. 'a' is zero-filled from the right.
>>	shift right - c>>d is 'c' shifted to the right by 'd' places. 'c' is zero-filled from the left.
! or |	bitwise OR (either will do, ! is Motorola standard) c|d is each bit in 'c' ORed with each bit in 'd'
&	bitwise AND c&d is each bit in 'c' ANDed with each bit in 'd'

These operators may be used to combine values anywhere that an expression is allowed. The operators are evaluated according to the following precedence table:

tightest binding	monadic - (minus)
then	<< >>
then	& !
then	* /
loosest binding	+ -

With equal precedence, evaluation is from left to right. Brackets may be used to change this if required.

If you wish to find out quickly whether or not a given expression is valid, it is very easy to 'try it out' using the P command.

4.6 Directives

The following Motorola compatible directives are supported by the assembler: EQU, SET, END, DS, DC, LIST, NOLIST, NOL, IFEQ, IFNE, ENDC, MACRO, MEXIT, ENDM. The assembler does not have any cross reference facilities. However, for compatibility reasons the directives PAGE, SPC, NOPAGE, LLEN, TTL, NOOBJ and G are ignored by the assembler and do not produce error messages. These directives are used by other assemblers to convey listing control inforation, and have no effect on the code produced by those assemblers.

In addition to those directives listed above, EQUR, CNOP, DISCARD, PATCH and GET have been added. The effect of each directive is now covered in more detail.

EQU and SET

EQU is used to declare a symbolic name for an absolute or relative value.

Example:

```
fred    EQU     23
joe     EQU     17+fred
```

SET is used to redefine a symbolic name. Note that the argument to SET should not contain forward references, or the results may be somewhat confusing.

Example:

```
pos     SET     0
...
pos     SET     pos+1
...
pos     SET     pos+2
```

EQUR

EQUR means 'equate register'. It is used to give a symbolic name to a register.

Example:

```
P       EQUR    A3
```

This allows 'P' to be used instead of A3. Thus, registers may be referred to by their use rather than by their number. This mechanism can also be used to give a symbolic name to any addressing mode.

Example:

```
NEXT    EQUR    (A3)
DATA    EQUR    4(A3)
```

This allows access to structures to be organised in a more helpful way.

DS

DS is used to reserve an area of storage. It takes a single argument, which is the number of storage units to be reserved. The default unit size is a word (16 bits), but .L or .B can be used to adjust this. The store units reserved will be aligned to a word boundary unless the size modifier is .B, when it will be aligned to a byte boundary. Zero bytes will be added beforehand if necessary, and if the instruction is labelled then the label indicates a position just after these extra bytes.

Example:

```
array: DS.L    asize    reserves asize long words of store, and defines
*                       "array" as a relative symbol which points at
*                       the first one
```

DS 0 can be used to align memory to a 2 byte boundary.

Note that the precise behaviour of DS and DC on correcting alignment of data objects may vary slightly between different 68000 assemblers.

DC

DC is used to reserve an initialised area of store. It may take any number of arguments. Each one is used to initialise a storage unit (a word by default, but can be changed using .B or .L). An exception to this is that if an argument is a charcter constant, then it may contain any number of characters and they will be placed in consecutive bytes of storage. Each entry made from a DC is aligned to a byte or word boundary, however, so that extra bytes may be inserted to retain the alignment.

The store units allocated will be aligned on a 1 or 2 byte boundary depending on the size modifier of the directive. Zero bytes will be added beforehand if necessary. If the instruction is labelled then the label indicates a position just after these extra bytes.

Examples:

```
        DC.L    3       aligns to a word boundary, then leaves
*                       a long word containing the value 3
        DC.B    'hello' leaves 5 bytes
```

CNOP

This 'Conditional NOP' directive is an extension from the Motorola standard and allows a section of code to be aligned on any boundary. Such a facility can be useful in certain types of programming applications.

The first argument is an offset, while the second argument is the alignment required for the base. The code will be aligned to the specified offset from the nearest required alignment boundary.

Syntax: CNOP offset,alignment

Example:
```
        CNOP    0,4     (align to nearest long word boundary)
        CNOP    2,4     (align to word boundary 2 bytes after the
                         nearest long word boundary)
```

LIST, NOLIST and NOL

These listing control directives allow different parts of an assembly listing to be printed out. All text following a NOL or NOLIST directive is not be printed. Printing recommences when a LIST directive is encountered. Text after a LIST directive will be printed out. The LIST option is selected by default.

END

This directive indicates the end of the assembly. It is not necessary, and is implied by the end of the input text file. If encountered in the middle of a file, assembly will terminate at that point.

This can be useful for trying out very short pieces of code. With the main body of text still in the memory, short pieces of code can be inserted at the top of the text file, followed by end. This is especially useful when used in conjunction with 'patch'. The assembler will then only assemble this short piece of code and allow it to be debugged effectively.

4.7 MACROs

Programs often contain coding of a repeated pattern of instructions which within themselves contain variable entries each time the code is used. Macros provide a shorthand notation for handling these repeated patterns within a program. Once a programmer has recognised a repeated pattern, he can designate any fields within the macro as variable. Once set up, the macro can be invoked many times over, substituting different parameters for the variable portions of the statements.

When the macro is defined, it is given a label. This label becomes the mnemonic by which the macro is subsequently called. To avoid confusion, the name of a macro definition must not be the same as an existing instruction or assembler directive.

An example of a typical MACRO definition might be:

```
*
*               A typical MACRO example
*
TRP     MACRO
        MOVEQ   #\2,D0
        TRAP    #\1
        ENDM
*
*               End of MACRO definition
*
```

This particular example would add the special MACRO called TRP. Two arguments would be required, so

```
        TRP     1,0
```

would cause the instructions:

```
        MOVEQ   #0,D0
        TRAP    #1
```

to be inserted into the text of the assembling program. The advantage is clear. Several mnemonics have been replaced by one. For larger pieces of code, the savings are even greater. We will now look at how this works.

MACRO definition

Each MACRO is defined by a label followed by MACRO. In the example above, this label was 'TRP'. Subsequent uses of this label as an operand will cause the contents of the macro to be expanded and inserted in the source code. The end of the MACRO is then defined by ENDM. Alternatively, the MACRO can be exited prematurely using the directive MEXIT. This would normally be used within the conditional assembly directives IFEQ or IFNE and ENDC.

When a macro name is used, it can be followed by a number of arguments, separated by commas. These arguments can then be inserted into the macro definition by the use of \ followed by the argument number. Hence, \1 and \2 in the example were replaced by arguments one and two. Up to ten such arguments can be used. Note that \0 has the special meaning of the size modifier (if any) appended to the macro label.

When an argument's value has multiple parts, or contains spaces, the value must be enclosed within angled brackets. The bracketed value must still be enclosed by the conventional commas. If a null argument is required, the argument can simply be left out, but the commas must still be provided to delimit the argument's position.

It is often useful to include labels within a macro definition, for example in a simple counting loop. However, since macros can be used over and over again, if the label were the same in each case then multiple label errors would be generated. This problem can be avoided by using assembler generated labels in the label field. Such labels have the form "@nnn" where "nnn" is a decimal number from 001 to 999 inclusive. The assembler will generate a new number for each new macro expansion. Subsequent labels encountered in subsequent macro expansions will therefore be unique.

Within the macro definition, \@ is used to represent such an assembler generated label.

The following example illustrates many of the points raised above:

```
EXAM    MACRO
        ADD     \1
\@      NOP
        DBRA    D0,\@
        DC.L    '\2MM\3'
        MEXIT
```

this generates the following expansions:

```
        EXAM    <D3,D4>,,NN
        ADD     D3,D4
@001    NOP
        DBRA    D0,@001
        DC.L    'MMNN'


        EXAM    <VAR-3,D2>,L,N
        ADD     VAR-3,D2
@002    NOP
        DBRA    D0,@002
        DC.L    'LMMN'
```

4.8 Conditional assembly

When producing large programs, it is often useful to be able to assemble certain parts only for debugging purposes, or only for a particular I/O application. Conditional assembly allows such alternatives to be selected by setting a flag to a particular value before assembly.

IFEQ and IFNE are the directives which enable or disable assembly, depending on the value of the expression in the operand field. The value is EQUAL if it is zero, and NOT EQUAL otherwise. The conditional assembly remains in force until the ENDC operand is encountered, at which point normal assembly resumes. It is possible to nest conditional assembly statements arbitrarily, but each level of nesting must be terminated by a matching ENDC.

The macro MEXIT directive is often used in conjunction with conditional statements to terminate the generation of a macro. For example,

```
TRY     MACRO
        MOVEQ   #56,D0
        IFNE    \1-0    IS THE ADD REQUIRED?
        ADDQ    #\1,D1
        MEXIT           DON'T DO THE SUBTRACT
        ENDC
        SUBQ    #1,D1
        ENDM
```

The above macro will add the argument to D1 if non-zero, or subtract 1 from D1 if the argument is zero, so

```
        TRY     4       adds 4 to D1

but     TRY     0       subtracts 1 from D1
```

4.9 PATCH and debugging large files

When very large files are being assembled and debugged, it can be very time consuming if the entire source file is re-assembled after every minor correction has been made.

For this reason, the ADDER assembler has been provided with a novel feature called patching. This procedure allows small changes to be made to an assembled file without re-assembling from the original source text. The operation of 'patch' is perhaps best illustrated by an example.

Example:

Original text in source file was:

```
        ...
        ...
        SUBQ    #1,D1
LABEL   MOVEQ   #1,D0
        TRAP    #1
        RTS
        ...
```

Debugging the program indicates that **TRAP #0** should have been used in place of **TRAP #1**. Instead of re-assembling the modified file, the following patch can be inserted at the front of the file:

```
PATCH   LABEL+2
TRAP    #0
END
...
```

When this piece of code is assembled, the binary code for TRAP #0 is inserted into the object code in place of the original code for TRAP #1. The file has been 'patched' and can be run for further debugging.

Several points should be noted. If the assembler operated in its normal mode during the above procedure, the symbol table which was generated on the first complete assembly of the source file would have been scrapped. The value for LABEL as used in the PATCH definition would not be known without re-assembling the entire file. This problem is overcome by LOCKing the symbol table after the first complete assembly. The **M** command is used for this. Once locked, the symbol table cannot be deleted until it is unlocked.

The symbol table should be kept in the locked state whilst PATCHing is in progress. As soon as too many patches are being used, it is advisable to update the relevant parts of the original source text and re-assemble. Before this operation, the symbol table must be UNLOCKed again using the **M** command. If this isn't performed, 'redefinition of a symbol' errors will be generated when assembly is restarted.

To summarise, the symbol table must be LOCKed using the **M** command before patching a file. The PATCH directive is followed by a label and optional offset (default of 0). This defines the position of the instructions which are to be corrected. The corrected instructions then follow the PATCH directive on subsequent lines in standard assembler format. These are terminated by the END directive. Assembly then modifies the object code for testing. When the entire file needs to be re-assembled again, the symbol table must be UNLOCKed.

4.10 Larger files and space management

Whilst the general philosophy of this assembler development package has hinged around the ability to write and debug small programs, facilities for coping with larger programs also exist.

The assembler has a fairly complex space management scheme. The major problem with larger files is that the source text, symbol table, assembler, editor, assembled code etc. just don't fit into memory together. There are three areas of memory which could run out:

the stack - this should never happen
the nodeheap - caused by too many forward references
the heap - probably caused by too much code, too many symbols or too much text.

The amount of remaining heap, which is the main memory store, is constantly displayed on the screen.

The problem of running out of memory can be circumvented to a large extent by keeping large chunks of text on microdrive. For example, suppose that the main body of a program has been saved in the microdrive file 'mdv1_main'. We can use the assembler directive **GET** to load this section of text into our program residing in memory.

The form of this directive is:

```
GET     'filename'
```

The filename should be in single quotes, just like a character constant in DC. The named file is opened and included in the source of the assembly. It is through this mechanism that a source of several files can be read. GET directives can be nested to any depth.

Example:

```
GET     'mdv1_main'
```

The use of GET allows standard library files to be built up for certain common tasks. This means that definitions of standard functions do not have to be retyped whenever they are used.

Another method of saving memory is by economising on the use of the symbol table space. The **DISCARD** directive can be used to delete entries from the symbol table. This directive is non-standard for Motorola assemblers, which are mainly 'two pass assemblers'. The Adder assembler is a single pass assembler, so using discard is possible.

The form of this directive is:

```
DISCARD <list of variables>
```

Using discard allows large programs to be assembled without using all of the symbol table space, since local variables can be deleted when they are no longer required. It also allows local label names to be reused.

Example:

```
V1:     ...
        BRA     V1
        DISCARD V1,V2,fred
        ...
V1:     ...
        BEQ     V1
```

If the system still runs out of memory, try saving and rebooting the entire system, especially if it has been in use for an extended period. This may solve the problem, because the heap sometimes suffers from "fragmentation" whereby it is unable to use the available free space for organisational reasons. If after saving and rebooting the system still runs out of memory then you probably require an extra half-megabyte plug-in memory expansion!

Appendix A - Errors

The assembler quotes errors by an error number, and a suitable message. This appendix lists all possible error messages. All errors are remembered, and passed on to the editor so that it can move rapidly to the location of each error in turn. If twenty errors are found then assembly is aborted, as something is obviously very wrong.

Some of the errors also produce additional information, which may or may not be informative. Errors involving files (for instance, not being able to open a file) may produce a small negative number - this is the QDOS error code and can be looked up in QDOS documentation. It may be of some help in deciding what has gone wrong.

Sometimes the error message is not helpful, and merely implys that "something is wrong with this line". Do not panic! Follow this procedure:

(1) check that there are some spaces in front of the opcode, and NO SPACES IN THE ARGUMENTS (except in quoted literals).

(2) Check the instruction carefully in the microprocessor reference manual. Have you used it correctly? e.g., check which sizes are allowed, and which addressing modes.

(3) Check all the symbol definitions of symbols used in the operands. (it may be useful to leave the editor, reassemble, and use the P command).

(4) Correct all the other errors and reassemble - this one could have been caused by an earlier error.

Assembler Error Codes and their Associated Meanings:

1 run out of heap space
    too many symbols, or too much text held in memory
2 run out of stack space
    shouldn't ever happen
3 data error
4 run out of node space
    too many symbols, or too many forward references
5 internal error
6 unexpected end of input
7 error from filing system
    frequently caused by a file not being found
    frequently followed by a negative number, which is an
    error code from QDOS
8 end of file in macro body
    A macro body may not span more than one file
    Probably caused by missing ENDM at end of a macro definition
9 badly formed size modifier
    a size modifier is of the form .L, .W, or .B for data values
    and .S for short branches. No other form is allowed
10 no closing quote in string/character constant
11 character constant with no characters in it

```
12  misplaced #
        # is used to denote immediate data
        and must be followed by a constant absolute expression
13  leading - not followed by (An) or expression
14  bad object in bracketed expression
15  argument missing or garbage found
16  misplaced <
17  misplaced >
18  bad object found in bracketed expression
19  size modifier on bogus object?
20  bogus address register in a(An) form?
21  address mode malformed
22  bad index modifier, or comma or bracket expected
23  comma or close bracket expected in index mode
24  bad operator arguments
25  subtract with bad arguments
26  register range mismatch
        reg-reg argument form (only legal as argument to MOVEM)
        used with two inappropriate registers
27  garbage found after arguments
        arguments must be separated from comments by spaces or tabs
28  wrong number of arguments for this opcode
29  close bracket expected
30  opcode not recognised
        The opcode on this line is not accepted by this assembler
        Check that there is space before and after it on the line
        Opcode case is not significant, and bad usage or size
        modifiers will not cause this error.
31  absolute value needed
        sometimes caused by illegal use of
        the program counter relative addressing mode
32  mismatched arguments for plus
        the only operations allowed for plus are:
           abs + abs (yielding abs)
           rel + abs (yielding rel)
           abs + rel (yielding rel)
33  mismatched arguments for minus
34  mismatched arguments for minus
35  attempt to divide by zero in constant calculation
36  shift count or ADDQ/SUBQ argument won't fit
        the value must be in the range 1 ..8
37  even address expected
        usually caused by DC.B yielding an odd number of bytes,
        followed by instructions or word/long word data.
38  even address expected
39  internal SA expansion error
40  the value must be in the range -32768..32767
41  the value must be in the range -32768..65535
42  the value must be in the range -128..127
43  offset of zero in short branch not allowed
44  the value must be in the range -128..255
45  redefinition of a symbol
        usually caused by a label being used twice
46  bad argument for EQU
47  EQU with no label
48  bad argument for SET
49  SET with no label
50  EQUR with no label
```

51 EQUR symbol has other definitions
EQU and EQUR may not be used on the same symbolic name
52 bad argument for DISCARD
53 bad argument for DISCARD
DISCARD should be given a list of names, as previously been defined using SET, EQU or EQUR
54 bad object as DC argument
55 not enough arguments for UPDATE operation
56 bad object as UPDATE argument
57 SR may not be UPDATEd, only CCR
58 too many arguments for UPDATE operation
59 data value expected
60 bad argument for GET
61 bad argument for GET
62 bad argument for PRINT command
63 bad address argument
64 bad arguments for DUMP operation
65 DUMP must start at an even address
66 this addressing mode cannot be used for this operation
Check the user's manual for exactly how to use this instruction - frequently caused by writing to a program relative location or some similar mistake.
67 bad use of PC mode
PC may only be used in the form label(PC) or value(PC)
68 bad argument for branch instruction
69 bad second (destination) argument to DBcc instruction
70 too many arguments for shift operation
71 bad arguments for register shift operation
72 bad first argument for bit operation
73 the given size specifier is not allowed for this operation
check the assembler manual for exactly what combination of arguments is allowed for the instruction on this line
74 move to USP is only allowed from an address register
75 move from USP is only allowed to an address register
76 a byte sized value may not be moved to an address register
an address register can only be a destination in a word or long word move
77 the first argument to MOVEQ must be immediate data
78 the second argument to MOVEQ must be a data register
79 MOVEP only works with word or long word arguments
80 MOVEP is only allowed to or from a data register
81 MOVEP needs address register with displacement addressing mode
82 MOVEM expects a register list argument
83 MOVEM.B not allowed
84 bad args for ABCD/SBCD
85 bad args for ABCD/SBCD
86 CMP must be to a data register
87 bad arguments for ADD/SUB/AND/OR/EOR
ADD/SUB/AND/OR must be to or from a data register
EOR must be from a data register
88 ADDA/SUBA/CMPA not to an address register
89 ADDA.B not allowed
90 immediate opcode without immediate first operand
91 immediate data expected as first argument
92 ADDQ.B/SUBQ.B not allowed to address register
93 bad arguments for CMPM
94 second argument of CHK/MUL/DIV must be a data register

```
95   EXG only allowed between registers
96   EXG only allowed between registers
97   EXT may only take a data register argument
98   EXT.B not allowed
99   SWAP may only take a data register argument
100   first argument to LINK must be an address register
101   second argument to LINK must be a 16 bit immediate data value
102   LEA second argument must be an address register
103   size modifier on TRAP not allowed
104   TRAP argument must be in the range 0..15
105   UNLK expects address register argument
106   MACRO found in macro body
         Nested macro definitions are not allowed.
         This error is usually caused by forgetting ENDM
         at the end of the previous macro definition.
107   macro definition redefines name
108   bad macro expansion key
         Somewhere in the body of the macro called on this line
         is a \ not followed by a digit or an @ sign.
109   garbage found after macro arguments
         arguments must be separated from comments
         by spaces or tabs
110   unclosed argument bracket in macro call
```

Appendix B – 68008 Instruction Set Summary

Instruction Set

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

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

Variations of Instruction Types

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

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