ASSEMBLER DEVELOPMENT KIT FOR THE QL

Preface

Chapter 1: The Screen Editor

Metacomco's Assembler Development Kit for the QL is a powerful package incorporating a full screen editor and a macro assembler. This book is intended to be a guide for users of the kit and does not aim to be fully comprehensive on all related aspects of the QL or assembler programming. It assumes that the reader has knowledge of the QDOS operating system.

If further detailed information is required, a full specification of the Motorola 68008 microprocessor can be found in *MC68000 16/32 Bit Microprocessor Programmer's Reference Manual* (4th edition, ISBN 1-356-6795X) published by Prentice-Hall. An introduction to assembler programming can be found in various introductory texts such as *Programming the M68000* by Tim King and Brian Knight (ISBN 0-201-14635-5) published by Addison-Wesley. Further information about QDOS can be found in *QL Advanced User Guide* by Adrian Dickens (ISBN 0-947929-00-2) published by Adder Publishing.

Assembler Development Kit for the QL

Contents

Chapter 1: The Screen Editor

1.1 Introduction
1.2 Immediate Commands
1.3 Extended Commands
1.4 Command List

Chapter 2: The Macro Assembler

2.1 Introduction
2.2 Running the Assembler
2.3 Program Encoding
2.4 Expression
2.5 Addressing Modes
2.6 Directives
2.7 Example Programs

Appendix A: The 68000 Instruction Set

Appendix B: Changing the Default Window

Appendix C: The Linker Program

1.1 Introduction

The screen editor ED may be used to create a new file or to alter an existing one. The text is displayed on the screen, and can be scrolled vertically or horizontally as required. The size of the program is about 20K bytes and it requires a minimum workspace of 8K bytes.

The editor is invoked using EXEC or EXEC_W as follows

```
EXEC_W mdv1_ed
```

The difference between invoking a program with EXEC or EXEC_W is as follows. Using EXEC_W means that the editor is loaded and SuperBasic waits until the editing is complete. Anything typed while the editor is running is directed to the editor. When the editor finishes, keyboard input is directed at SuperBasic once more.

Using EXEC is slightly more complicated but is more flexible. In this case the editor is loaded into memory and is started, but SuperBasic carries on running. Anything typed at the keyboard is directed to SuperBasic unless the current window is changed. This is performed by typing CTRL-C, which switches to another window. If just one copy of ED is running then CTRL-C will switch to the editor window, and characters typed at the keyboard will be directed to the editor. A subsequent CTRL-C switches back to SuperBasic. When the editor is terminated a CTRL-C will be needed to switch back to SuperBasic once more. More than one version of the editor can be run concurrently (subject to available memory) if EXEC is used. In this case CTRL-C switches between SuperBasic and the two versions of the editor in turn.

Once the program is loaded it will ask for a filename which should conform to the standard QDOS filename syntax. No check is made on the name used, but if it is invalid a message will be issued when an attempt is made to write the file out, and a different file name may be specified then if required. All subsequent questions have defaults which are obtained by just pressing ENTER

The next question asks for the workspace required. ED works by loading the file to be edited into memory and sufficient workspace is needed to hold all the file plus a small overhead. The default is 12K bytes which is sufficient for small files. The amount can be specified as a number or in units of 1024 bytes if the number is terminated by the character K. If you ask for more memory than is available then the question is asked again. The minimum is 8K bytes.

You are next asked if you wish to alter the window used by ED. The default window is normally the same as the window used in the initialisation of ED although this may be altered if required. See Appendix B for details of how to do this. If you type N or just press ENTER then the default window is used. If you type Y then you are given a chance to alter the window. The current window is displayed on the screen and the cursor keys can be used to move the window around. The combination ALT and the cursor keys will alter the size of the window although there is a minimum size which may be used. Within this constraint you can specify a window anywhere on the screen, so that you can edit a file and do something else such as run a SuperBasic program concurrently. When you are satisfied with the position of the window press ENTER.

Next, an attempt is made to open the file specified, and if this succeeds then the file is read into storage and the first few lines displayed on the screen. Otherwise a blank screen is provided, ready for the addition of new data. The message "File too big" indicates that more workspace should be specified.

When the editor is running the bottom line of the screen is used as a message area and command line. Any error messages are displayed there, and remain displayed until another editor command is given.

Editor commands fall into two categories - immediate commands and extended commands. Immediate commands are those which are executed immediately, and are specified by a single key or control key combination. Extended commands are typed in onto the command line, and are not executed until the command line is finished. A number of

extended commands may be typed on a single command line, and any commands may be grouped together and groups repeated automatically. Most immediate commands have a matching extended version.

Immediate commands use the function keys and cursor keys on the QL in conjunction with the special keys SHIFT, CTRL and ALT. For example, delete line is requested by holding down the CTRL and ALT keys and then pressing the left arrow key. This is described in this document as CTRL-ALT-LEFT. Function keys are described as F1, F2 etc.

The editor attempts to keep the screen up to date, but if a further command is entered while it is attempting to redraw the display, the command is executed at once and the display will be updated later, when there is time. The current line is always displayed first, and is always up to date.

1.2 Immediate commands

Cursor control

The cursor is moved one position in either direction by the cursor control keys LEFT, RIGHT, UP an DOWN. If the cursor is on the edge of the screen the text is scrolled to make the rest of the text visible. Vertical scroll is carried out a line at a time, while horizontal scroll is carried out ten characters at a time. The cursor cannot be moved off the top or bottom of the file, or off the left hand edge of the text.

The ALT-RIGHT combination will take the cursor to the right hand edge of the current line, while ALT-LEFT moves it to the left hand edge of the line. The text will be scrolled horizontally if required. In a similar fashion SHIFT-UP places the cursor at the start of the first line on the screen, and SHIFT-DOWN places it at the end of the last line on the screen.

The combinations SHIFT-RIGHT and SHIFT-LEFT take the cursor to the start of the next word or to the space following the previous word respectively. The text will be scrolled vertically or horizontally as required. The TAB key can also be used. If the cursor position is beyond the end of the current line then TAB moves the cursor to the next tab position, which is a multiple of the tab setting (initially 3). If the cursor is over some text then sufficient spaces are inserted to align the cursor with the next tab position, with any characters to the right of the cursor being shuffled to the right.

Inserting text

Any letter typed will be added to the text in the position indicated by the cursor, unless the line is too long (there is a maximum of 255 characters in a line). Any characters to the right of the text will be shuffled up to make room. If the line exceeds the size of the screen the end of the line will disappear and will be redisplayed when the text is scrolled horizontally. If the cursor has been placed beyond the end of the line, for example by means of the TAB or cursor control keys, then spaces are inserted between the end of the line and any inserted character. Although the QL keyboard generates a different code for SHIFT-SPACE and SHIFT-ENTER these are mapped to normal space and ENTER characters for convenience.

An ENTER key causes the current line to be split at the position indicated by the cursor, and a new line generated. If the cursor is at the end of a line the effect is simply to create a new, empty blank line after the current one. Alternatively CTRL-DOWN may be used to generate a blank line after the current, with no split of the current line taking place. In either case the cursor is placed on the new line at the position indicated by the left margin (initially column one).

A right margin may be set up (using the command SR) so that ENTERs are automatically inserted before the preceding word when the length of the line being typed exceeds that margin. In detail, if a character is typed and the cursor is at the end of the line and at the right margin position then an automatic newline is generated. Unless the character typed was a space, the half completed word at the end of the line is moved down to the newly generated line. (Note that if a line has no spaces, i.e. it is completely full of characters, the whole line of text is considered to be one 'word' which is then moved down to a new line, leaving an empty line above). Initially there is a right margin set up at the right hand edge of the window used by ED. The right margin may be disabled by means of the EX command (see later).

Deleting text

The CTRL-LEFT key combination deletes the character to the left of the cursor and moves the cursor left one position. If the cursor is at the start of a line then the newline between the current line and the previous is deleted (unless you are on the very first line). The text will be scrolled if required. CTRL-RIGHT deletes the character at the current cursor position without moving the cursor. As with all deletes, characters remaining on the line are shuffled down, and text which was invisible beyond the right hand edge of the screen may now become visible.

The combination SHIFT-CTRL-RIGHT may be used to delete a word or a number of spaces. The action of this depends on the character at the cursor. If this character is a space then all spaces up to the next non-space character on the line are deleted. Otherwise characters are deleted from the cursor, and text shuffled left, until a space is found. The CTRL-ALT-RIGHT command deletes all characters from the cursor to the end of the line. The CTRL-ALT-LEFT command deletes the entire current line.

Scrolling

Besides the vertical scroll of one line obtained by moving the cursor to the edge of the screen, the text may be scrolled 12 lines vertically by means of the commands ALT-UP and ALT-DOWN. ALT-UP moves to previous lines, moving the text window up; ALT-DOWN moves the text window down moving to lines further on in the file. The F4 key rewrites the entire screen, which is useful if the screen is altered by another program besides the editor. Remember that you can switch out of the editor window and into some other job by typing CTRL-C at any point, assuming that there is another job with an outstanding input request. SuperBasic will be available only if you entered the editor using EXEC rather than EXEC_W. If there is enough room in memory you can run two versions of ED at the same time if you wish.

Repeating commands

The editor remembers any extended command line typed, and this set of extended commands may be executed again at any time by simply pressing F2. Thus a search command could be set up as the extended command, and executed in the normal way. If the first occurrence found was not the one required, typing F2 will cause the search to be executed again. As most immediate commands have an extended version, complex sets of editing commands can be set up and executed many times. Note that if the extended command line contains repetition counts then the relevant commands in that group will be executed many times each time the F2 key is pressed.

1.3 Extended commands

Extended command mode is entered by pressing the F3 key. Subsequent input will appear on the command line at the bottom of the screen. Mistakes may be corrected by means of CTRL-LEFT and CTRL-RIGHT in the normal way, while LEFT and RIGHT move the cursor over the command line. The command line is terminated by pressing ENTER. After the extended command has been executed the editor reverts to immediate mode. Note that many extended commands can be given on a single command line, but the maximum length of the command line is 255 characters. An empty command line is allowed, so just typing ENTER after typing F3 will return to immediate mode.

Extended commands consist of one or two letters, with upper and lower case regarded as the same. Multiple commands on the same command line are separated from each other by a semicolon. Commands are sometimes followed by an argument, such as a number or a string. A string is a sequence of letters introduced and terminated by a delimiter, which is any character besides letters, numbers, space, semicolon or brackets. Thus valid strings might be

```
/happy/   !23 feet!  :Hello!: "1/2"
```

Most immediate commands have a corresponding extended version. See the table of commands for full details (section 1.4).

Program control

The command X causes the editor to exit. The text held in storage is written out to file, and the editor then terminates. The editor may fail to write the file out either because the filename specified when editing started was invalid, or because the microdrive becomes full. In either case the editor remains running, and a new destination should be specified by means of the SA command described below. Alternatively the Q command terminates immediately without writing the buffer; confirmation is requested in this case if any changes have been made to the file. A further command allows a 'snapshot' copy of the file to be made without coming out of ED. This is the SA command. SA saves the text to a named file or, in the absence of a named file, to the current file. For example:

```
*SA /mdv2_savedtext/
```

or

```
*SA
```

This command is particularly useful in areas subject to power failure or surge. It should be noted that SA followed by Q is equivalent to the X command. Any alterations made between the SA and the Q will cause ED to request confirmation again; if no alterations have been made the program will be quitted immediately with the file saved in that state. SA is also useful because it allows the user to specify a filename other than the current one. It is therefore possible to make copies at different stages and place them in different files.

The SA command is also useful in conjunction with the R command. Typing R followed by a filename causes the editor to be re-entered editing the new file. The old file will be lost when this happens, so confirmation is requested (as with the Q command) if any changes to the

current file have been made. The normal action is therefore to save the current file with SA, and then start editing a new file with R. This saves having to load the editor into memory again, and means that once the editor is loaded the microdrive containing it can be replaced by another.

The U command "undoes" any alterations made to the current line if possible. When the cursor is moved from one line to another, the editor takes a copy of the new line before making any changes to it. The U command causes the copy to be restored. However the old copy is discarded and a new one made in a number of circumstances. These are when the cursor is moved off the current line, or when scrolling in a horizontal or vertical direction is performed, or when any extended command which alters the current line is used. Thus U will not "undo" a delete line or insert line command, because the cursor has been moved off the current line.

The SH command shows the current state of the editor. Information such as the value of tab stops, current margins, block marks and the name of the file being edited is displayed. Tabs are initially set at every three columns; this can be changed by the command ST, followed by a number n, which sets tabs at every n columns. The left margin and right margin can be set and reset by SL and SR commands, again followed by a number indicating the column position. The left margin should not be set beyond the width of the screen. The EX command may be used to extend margins; once this command is given, no account will be taken of the right margin on the current line. Once the cursor is moved off the current line, margins are enabled once more. To extend more than just the current line, use the SR command. The right hand margin may be set beyond the width of the window or screen. The maximum width is 255. (See also 1.2 'Inserting text').

Block control

A block of text can be identified by means of the BS (block start) and BE (block end) commands. The cursor should be moved to the first line required in a block, and the BS command given. The cursor can then be moved to the last line wanted in the block, by cursor control commands or in any other way, such as searching. The BE command is then used to mark the end of the block. Note, however, that if any change is made to the text the block start and block end become undefined once more. The start of the block must be on the same line, or a line previous to, the line which marks the end of the block. A block always contains all of the line(s) within it.

Once a block has been identified, a copy of it may be moved into another part of the file by means of the IB (insert block) command. The previously identified block is replicated immediately after the current line. Alternatively a block may be deleted by means of the DB command, after which the block start and end values are undefined. It is not possible to insert a block within itself.

Block marks may also be used to remember a place in a file. The SB (show block) command resets the screen window on the file so that the first line in the block is at the top of the screen.

A block may also be written to a file by means of the WB command. The command is followed by a string which represents a file name. The file is created, possibly destroying the previous contents, and the buffer written to it. A file may be inserted by the IF command. The filename given as the argument string is read into storage immediately following the current line.

Movement

The command T moves the screen to the top of the file, so that the first line in the file is the first line on the screen. The B command moves the screen to the bottom of the file, so that the last line in the file is the bottom line on the screen if possible.

The commands N and P move the cursor to the start of the next line and previous line respectively. The commands CL and CR move the cursor one place to the left or one place to the right, while CE places the cursor at the end of the current line, and CS places it at the start.

It is common for programs such as compilers and assemblers to give line numbers to indicate where an error has been detected. For this reason the command M is provided, which is followed by a number representing the line number which is to be located. The cursor will be placed on the line number in question. Thus M1 is the same as the T command. If the line number specified is too large the cursor will be placed at the end of the file.

Searching and Exchanging

Alternatively the screen window may be moved to a particular context. The command F is followed by a string which represents the text to be located. The search starts at one place beyond the current cursor position and continues forwards through the file. If found, the cursor is placed at the start of the located string. To search backwards through the text use the command BF (backwards find) in the same way as F. BF will find the last occurrence of the string before the current cursor position. To find the earliest occurence use T followed by F; to find the last, use B followed by BF. The string after F and BF can be omitted; in this case the string specified in the last F, BF or E command is used. Thus

```
F /wombat/

BF
```

will search for 'wombat' in a forwards direction and then in a reverse direction.

The E (exchange) command takes a string followed by further text and a further delimiter character, and causes the first string to be exchanged to the last. So for example

```
E /wombat/zebra/
```

would cause the letters 'wombat' to be changed to 'zebra'. The editor will start searching for the first string at the current cursor position, and continues through the file. After the exchange is done the cursor is moved to after the exchanged text. An empty string is allowed as the search string, specified by two delimiters with nothing between them. In this case the second string is inserted at the current cursor position. No account is taken of margin settings while exchanging text.

A variant on the E command is the EQ command. This queries the user whether the exchange should take place before it happens. If the response is N then the cursor is moved past the search string. If the response is Y or ENTER then the change takes place; any other response (except F2) will cause the command to be abandoned. This command is normally only useful in repeated groups; a response such as Q can be used to exit from an infinite repetition.

All of these commands normally perform the search making a distinction between upper and lower case. The command UC may be given which causes all subsequent searches to be made with cases equated. Once this command has been given then the search string "wombat" will match "Wombat", "WOMBAT", "WoMbAt" and so on. The distinction can be enabled again by the command LC.

Altering text

The E command cannot be used to insert a newline into the text, but the I and A commands may be used instead. The I command is followed by a string which is inserted as a complete line before the current line. The A command is also followed by a string, which is inserted after the current line. It is possible to add control characters into a file in this way.

The S command splits the current line at the cursor position, and acts just as though an ENTER had been typed in immediate mode. The J command joins the next line onto the end of the current one.

The D command deletes the current line in the same way as the CTRL-ALT-LEFT command in immediate mode, while the DC command deletes the character at the cursor in the same way as CTRL-RIGHT.

Repeating commands

Any command may be repeated by preceding it with a number. For example,

```
4 E /slithy/brillig/
```

will change the next four occurrences of 'slithy' to 'brillig'. The screen is verified after each command. The RP (repeat) command can be used to repeat a command until an error is reported, such as reaching the end of the file. For example,

```
RP E /slithy/brillig/
```

will change all occurrences of 'slithy' to 'brillig'.

Commands may be grouped together with brackets and these command groups executed repeatedly. Command groups may contain further nested command groups. For example,

```
RP ( F /bandersnatch/; 3 (IB; N))
```

will insert three copies of the current block whenever the string 'bandersnatch' is located.

Note that some commands are possible, but silly. For example,

```
RP SR 60
```

will set the right margin to 60 ad infinitum. However, any sequence of extended commands, and particularly repeated ones, can be interrupted by typing any character while they are taking place. Command sequences are also abandoned if an error occurs.

1.4 Command list

In the extended command list, /s/ indicates a string, /s/t/ indicates two exchange strings and n indicates a number.

Immediate commands

F2	Repeat last extended command
F3	Enter extended mode
F4	Redraw screen
LEFT	Move cursor left
SHIFT-LEFT	Move cursor to previous word
ALT-LEFT	Move cursor to start of line
CTRL-LEFT	Delete left one character
CTRL-ALT-LEFT	Delete line
RIGHT	Move cursor right
SHIFT-RIGHT	Move cursor to start of next word
ALT-RIGHT	Move cursor to end of line
CTRL-RIGHT	Delete right one character
CTRL-ALT-RIGHT	Delete to end of line
SHIFT-CTRL-RIGHT	Delete word to right
UP	Move cursor up
SHIFT-UP	Cursor to top of screen
ALT-UP	Scroll up
DOWN	Move cursor down
SHIFT-DOWN	Cursor to bottom of screen
ALT-DOWN	Scroll down
CTRL-DOWN	Insert blank line

Extended Commands

A /s/	Insert line after current
B	Move to bottom of file
BE	Block end at cursor
BF	Backwards find
BS	Block start at cursor
CE	Move cursor to end of line
CL	Move cursor one position left
CR	Move cursor one position right
CS	Move cursor to start of line
D	Delete current line
DB	Delete block
DC	Delete character at cursor
E /s/t/	Exchange s into t
EQ /s/t/	Exchange but query first
EX	Extend right margin
F /s/	Find string s
I /s/	Insert line before current
IB	Insert copy of block
IF /s/	Insert file s
J	Join current line with next
LC	Distinguish between upper and lower case in searches
M n	Move to line n
N	Move cursor to start of next line
P	Move cursor to start of previous line
Q	Quit without saving text
R /s/	Re-enter editor with file s
RP	Repeat until error
S	Split line at cursor
SA /s/	Save text to file
SB	Show block on screen
SH	Show information
SL n	Set left margin
SR n	Set right margin
ST n	Set tab distance
T	Move to top of file
U	Undo changes on current line
UC	Equate U/C and l/c in searches
WB /s/	Write block to file s
X	Exit, writing text back

Chapter 2: The Macro Assembler

2.1 Introduction

The Sinclair QL contains a Motorola 68008 microprocessor which was designed to supersede the MC6800 and MC6809 processors. It is a byte-addressed machine with a 20 bit address bus, giving it an address space of 1 megabyte. It has at any one time 16 32-bit registers, 8 of which are specialised for arithmetic type operations (data registers, named D0 - D7), and the other 8 of which are specialised for addressing operations (address registers, named A0 - A7). A7 (synonymous to SP) is designated as the Stack Pointer.

It has two operating states (User and System), the second of which is privileged, and has its own, separate stack pointer. This means that there are in fact 9 address registers, only 8 of which are visible at any one time. In addition to the general purpose registers, there is a 24-bit Program Counter (PC), and a 16-bit Status Register (SR) incorporating the 8-bit Condition Code register (CCR).

Like the MC6809, the MC68008 has a large collection of addressing modes, and addresses are specified by using one of the 12 Effective Address types, the assembler code syntax of which is described later. The MC68008 is capable of performing 8, 16 and 32 bit arithmetic, and most instructions have a size qualifier, specifying the size of the operation to be performed.

2.2 Running the Assembler

The assembler consists of three overlays, and will normally require most of the available memory of an unexpanded QL. Thus it is not possible to run the assembler with a large SuperBasic program or the editor in memory unless the QL has expansion RAM fitted. It may be possible to load the first assembler overlay with little spare RAM, but the assembler will not be able to load the second (and largest) overlay in this case. It is run using the EXEC or EXEC_W command as follows:

```
EXEC_W mdv1_asm
```

See the description of the editor for a fuller discussion of the difference between EXEC and EXEC_W. Once the program has loaded it will ask for the name of the source file. This file should contain a suitable assembly language program. This will consist of a mixture of instruction opcodes and directives, symbolic names and labels, operators and other special symbols. This file must be provided and must exist. Note that the assembler will require access to the microdrive from which it was loaded whilst it runs, so the normal course of action is to keep the source on one microdrive and the assembler on another. The assembler can be loaded from either drive.

The assembler then proceeds to ask a number of further questions, all of which have a default value which is used if just ENTER is pressed. The first of these is to determine whether an assembly listing is required, and if so where it is to be produced. The initial question asks if a listing is needed; the default is N which means that no listing will be produced. In this case if any error is detected the line in error and the error message will be displayed in the window. If the reply is Y then a further question is asked concerning the destination for the listing. The default is to produce the listing on the screen, and if ENTER is pressed then this is assumed. When the listing is produced on the screen, then no page headings are printed. If a file name is specified then this will be used. The file may refer to a microdrive file or to a serial line device such as ser1 to which a printer is connected. In this case page headings are produced at the top of each page, and directives such as PAGE (see below) will cause a page throw. Note that in order to copy a listing saved on microdrive to a printer COPY_N rather than just COPY should be used.

Assembly listings contain all the lines in the input file which are not disabled from being printed by either a NOLIST directive or because the line contains a directive which is not echoed. The output provides the address or relative offset followed by the code generated. This is followed by a space or an E if the line is in error. The line number and the reflection of the input text makes up the rest of the line. If the line is the result of a macro expansion then the line number will be followed by a plus sign ('+'). If there are any errors then a new page is given at the end of the listing detailing the error, with a full English error message and a reference to the line number.

The next question concerns the code file. This will contain the object code produced by the assembler. The default is that no code output is required, so none will be produced; this is useful if a simple check on the syntax of a program is required, or perhaps just an assembly listing is needed. If a file name is given then an attempt is made to open it, and the question is asked again if this fails.

The final stage is to determine whether the window used by the assembler is to be changed. The reply ENTER or N means that the default window is to be used. The default window is normally the same as the initial window, but this may be changed. See Appendix B for details. If Y is typed then the window is cleared and it may be moved or changed in size by means of the cursor keys. The cursor keys by themselves move the window, while ALT pressed in conjunction with the cursor keys will alter the window size. Once the window is suitably defined, ENTER will proceed with the assembly.

The output file may be in one of four different formats, depending on the assembler program. The formats may not be mixed. The first format is position independent code. In this case the output will be binary code which can be loaded anywhere in memory. In order to produce code which can be EXECed the file must be position independent, as there is no way to determine where in memory the code will be loaded. It is normally good style to write programs as position independent if at all possible. Code will be produced as position independent if ORG is not specified and if no relocation information is generated.

The second case is relocatable code. This is produced when ORG has not been specified but when relocation information is generated. Relocation information is produced when a reference is made to a label which cannot be computed while the assembler is running, but which will only be known when the program is loaded. For example,

```
        DC.L   FRED
        :
FRED    BSR.S  MARY
```

In this case the value of the label FRED cannot be specified relative to the program counter as MARY is, and the address represented by FRED will not be known until the program is loaded into memory. The reference to FRED requires relocation. This is performed by the assembler in the following fashion. The output file is produced in a special format which contains the code and the relocation information. This is preceded by a small section of code (a relocator) produced automatically from the assembler. When the code is run execution will start at the first instruction, which is part of the relocator. The relocator will use the relocation information to modify the program so the reference to FRED above does indeed refer to the instruction required. Once all such relocation information is dealt with, the first instruction of the user's code is jumped to. The effect of this is that relocatable code should execute as expected, but if the path of the code is traced with a debugger then the relocator will be seen to run before the main user program. Relocatable code can be run via EXEC or CALL.

The assembler can also produce code in absolute format. Absolute code will only run in one region of memory and will not work anywhere else. It is specified by using the ORG directive which is followed by a reference to the first location to be used. For example, space might be allocated via the RESPR SuperBasic command, and a note taken of the value returned. The program could then be assembled using the base of the resident area as the argument to the ORG directive. The resulting absolute code should be loaded into the resident area via LBYTES and CALLed. The program may not be EXECed and will not run at any other location. If possible absolute format should be avoided because of these restrictions. The two previous formats may be CALLed or EXECed and are hence much more flexible.

The final format is used if any external names are used, either as an external reference or an external definition. These will be used to link sections of code written in assembler or in high level languages. The output is not directly executable but must be passed through a linker to be combined with other code sections. This format is produced if XDEF or XREF are used.

The assembler is a two pass assembler, reading the source text twice, firstly to construct a symbol table and macro definitions, and secondly, to produce a source listing and object module.

Finally, once the assembler has finished, it asks

```
More files to assemble (Y/N) ?
```

If Y is replied, the whole series of questions at the beginning of this section is repeated. A reply of N ends assembly and takes the user out of the assembler. There is no default, i.e. a reply of ENTER leads to the question being repeated until Y or N is given. However, if the last assembly fails from a fatal error, the question is not asked. The assembler will not continue after a fatal error.

2.3 Program encoding

A program acceptable to the assembler takes the form of a series of input lines that are:

a) Comment or Blank lines
b) Executable Instructions
c) Assembler Directives

Comments

Comments are introduced into the program in one of two ways. If the first character on the line is an asterisk ("*"), then the whole line is treated as comment. Also, text after an instruction or assembler directive is treated as comment, provided it is preceded by at least one blank character. Blank lines are also treated as comments by the assembler.

Examples of Comments:

```
*  This entire line is a comment

FRED  MOVEQ.L  #10,D0  Comment following instruction

      PAGE             Move to the top of a new page
```

Executable Instructions

The specification of the executable instructions is given in full in the references provided in the preface and is not covered here. The source statements have the general overall format:

[LABEL] OPCODE [OPERAND(s)] [COMMENT]

each field being separated from the next by at least one blank character.

Label Field

A label is a user symbol which either

a) Starts in the first column, terminated by at least one blank character or a newline, or
b) Starts in any column, and is terminated with a colon (":").

If a label is present, then it must be the first non-blank item on the line. The label is assigned the value and type of the program counter, i.e. the memory address of the first byte of the instruction or data being referenced. Labels are allowed on all instructions, and some directives. See the specification of individual directives for whether a label field is allowed. Opcodes which are uses of a macro may not have a label field, although the same effect is obtained by placing the label by itself on the line before.

N.B. Labels must not be Instruction names, Directives or Register names, and must not be multiply defined.

Opcode Field

The Opcode field follows the Label field, and is separated from it by at least one blank character. Entries in this field are of two types. Firstly, the MC68000 operation codes, as defined in the MC68000 User Manual, and secondly, Assembler Directives. For instructions and directives which can operate on more than one data size, there is an optional Size-Specifier subfield, which is separated from the opcode by the period (".") character. Possible size specifiers are:

B - Byte sized data (8 bits)
W - Word sized data (16 bits)
L - Long Word sized data (32 bits)
    or Long Branch specifier
S - Short Branch specifier

The size specifier must match with the instruction or directive type being used.

Operand Field

If present, this field contains the one or more operands to the instruction or directive, and must be separated from it by at least one blank character. Where two or more operands occur in this field, they must be separated by the comma character (","). The operand field is terminated by the blank or newline characters, so no blank characters are allowed in this field.

Comment Field

If present, anything after the terminating blank character of the operand field is ignored, and hence can be treated as comment.

2.4 Expressions

An expression is a combination of symbols, constants, algebraic operators and parentheses, and can be used to specify the operand field to instructions or directives. Relative symbols may be included in expressions, but they can only be operated on by a subset of the operators.

Operators

The operators available, in decreasing order of precedence are:

a) Monadic Minus ("-")
b) Lshift, Rshift ("<<" and ">>")
c) And, Or ("&" and "!")
d) Multiply, Divide ("*" and "/")
e) Add, Subtract ("+" and "-")

The precedence of the operators can be over-ridden by enclosing sub-expressions in parentheses. Operators of equal precedence are evaluated from left to right. Note that there should not normally be any spaces in an expression, as the space will be regarded as a delimiter between one field and another.

Operand Types for Operators

In the following table, absolute symbols are represented by "A", and relative symbols by "R". The possible operator/operand combinations are shown, with the type of the resulting value. "x" indicates an error. The Monadic minus operator is only valid with an absolute operand.

Operators	Operands			
	A op A	R op R	A op R	R op A
+	A	x	R	R
-	A	A	x	R
*	A	x	x	x
/	A	x	x	x
&	A	x	x	x
!	A	x	x	x
>>	A	x	x	x
<<	A	x	x	x

Symbols

A symbol is a string of characters, the first of which must be alphabetic ("A"-"Z"), "@", or period (".") and the rest of which can be any of these characters or also numeric ("0"-"9") or the underline character ("_"). In all symbols, the lower case characters ("a"-"z") are treated as synonymous with their upper case equivalents. Symbols can be up to 30 characters in length, all of which are significant. The assembler will take symbols longer than this, and truncate them to 30 characters, giving a warning that it has done so. The Instruction names, Directive names, Register names, and special symbols CCR, SR, SP and USP cannot be used as user symbols. A symbol can have one of three types:

Absolute

a) The symbol was SET or EQUated to an Absolute value

b) An ORG statement preceded the definition of the symbol

Relative

a) The symbol was SET or EQUated to a Relative value

b) A RORG statement preceded the definition of the symbol

c) Neither ORG nor RORG was used (Default is RORG $0)

Register

a) The symbol was set to a register name using EQUR (This is an extension from the MOTOROLA specification).

There is a special symbol "*", which has the value and type of the current program counter, i.e. the address of the current instruction or directive being dealt with.

Numbers

A number may be used as a term of an expression, or as a single value. Numbers ALWAYS have absolute values, and can take one of the following formats:

Decimal (a string of decimal digits)

Example: 1234

Hexadecimal ("$" followed by a string of hex digits)

Example: $89AB

Binary ("%" followed by zeros and ones)

Example: %10110111

ASCII Literal (Up to 4 ASCII characters within quotes)

Examples: 'ABCD' 'e' 'Ian"s'

Strings of less than 4 characters are justified to the right, with NUL being used as the packing character.

To obtain a quote character in the string, two quotes must be used.

2.5 Addressing modes

The effective address modes define the operands to instructions and directives, and there is a detailed description of them in the references above. Addresses refer to individual bytes, but instructions, Word and Long Word references access more than one byte, and the address for these must be word aligned.

In the following table, "Dn" represents one of the data registers (D0-D7), "An" represents one of the address registers (A0-A7 and SP), "a" represents an absolute expression, "r" represents a relative expression, and "Xn" represents An or Dn, with an optional ".W" or ".L" size specifier. The syntax for each of the modes is as follows:

Address Mode	Description and Examples
Dn	Data Register Direct e.g. MOVE D0,D1
An	Address Register Direct e.g. MOVEA A0,A1
(An)	Address Register Indirect e.g MOVE D0,(A1)
(An)+	Address Register Indirect Post Increment e.g. MOVE (A7)+,D0
-(An)	Address Register Indirect Pre Decrement e.g. MOVE D0,-(A7)
a(An)	Address Register Indirect with Displacement e.g. MOVE 20(A0),D1

a(An,Xn)	Address Register Indirect with Index
	e.g. MOVE 0(A0,D0),D1
	MOVE 12(A1,A0.L),D2
	MOVE 120(A0,D3.W),D4

Address Mode	**Description and Examples**
a	Short absolute (16 bits) e.g. MOVE $1000,D0
a	Long absolute (32 bits) e.g. MOVE $10000,D0
r	Program Counter Relative with Displacement e.g. MOVE ABC,D0 (ABC is relative)
r(Xn)	Program Counter Relative with Index e.g. MOVE ABC(D0.L),D1(ABC is relative)
#a	Immediate data e.g. MOVE #1234,D0

USP)
CCR)
SR)
	Special addressing modes
	e.g. MOVE A0,USP
	MOVE D0,CCR
	MOVE D1,SR

2.6 Directives

All assembler directives (with the exception of DC) are instructions to the assembler, rather than instructions to be translated into object code. They are first listed by function, and then described individually.

Note that labels are only allowed on directives where shown. For example, EQU is allowed a label, it is optional for ORG but not allowed for LLEN or TTL.

Assembly Control

ORG	Absolute origin
RORG	Relocatable origin
SIZE	Size of data area
END	Program end

Symbol Definition

EQU	Assign permanent value
EQUR	Assign permanent register value
SET	Assign temporary value

Data Definition

DC	Define constants
DS	Define storage

Listing Control

PAGE	Page-throw to listing
LIST	Enable listing
NOLIST (NOL)	Disable listing
SPC n	Skip n blank lines
NOPAGE	Disable paging
LLEN n	Set line length (60 <= n <= 132)
PLEN n	Set page length (24 <= n <= 100)
TTL	Set program title (max 40 chars)
NOOBJ	Disable object code output
FAIL	Generate an assembly error

Conditional Assembly

CNOP	Conditional NOP for alignment
IFEQ	Assemble if equal
IFNE	Assemble if not equal
ENDC	End of conditional assembly

Macro Directives

MACRO	Define a macro name
ENDM	End of macro definition
MEXIT	Exit the macro expansion

External Symbols

XDEF	Define external name
XREF	Reference external name

General Directives

GET "<file>"	Insert file in the source

Assembly Control

ORG Set Absolute Origin

Format: [label] ORG absexp

The ORG directive changes the program counter to the value specified by the absolute expression in the operand field. Subsequent statements are assigned absolute memory locations, starting with the new program counter value. The optional symbol "label" will be assigned the value of the program counter AFTER the ORG directive has been obeyed.

RORG Set Relative Origin

Format: [label] RORG absexp

The RORG directive changes the program counter to be of relocatable type, and to have the value given by "absexp". Subsequent statements will be assigned relocatable memory locations, starting with the value assigned to the program counter. Addressing in relocatable sections is done using the "program counter relative with displacement" addressing mode.

If neither an ORG nor a RORG directive is specified, relative mode is assumed, and the program counter will start as if a "RORG 0" were present. Assembler modules which are to run via EXEC must be relocatable, so this default value is normally correct. The label value assignment is the same as for ORG.

SIZE Size of data area

Format: SIZE number

The SIZE directive tells the assembler the size of the data area to write into the code file, and which will hence be allocated to you when the job is run via EXEC. A job initiated in this way will be started with register A6 pointing to the base of the code, (A6,A4.L) will point to the base of the data area and (A6,A5.L) will point to the end of the data area. The stack pointer will normally run down from the top of the area towards the bottom; any other space can be allocated from the bottom upwards. Jobs should normally make use of the data area for memory storage by setting up suitable offsets on (A6,A4.L) or some other register initialised to this value. The default size of the data area if the SIZE directive is not given is 500 bytes. The number specified after the SIZE directive is the new value specified in bytes.

END End of program

Format: [label] END

The END directive tells the assembler that the source is finished, and subsequent source statements are ignored. The END directive encountered during the first pass of the assembler causes it to begin the second pass. If the end of file is detected before an END directive a warning message is given. If the label field is present, then the value of the current program counter is assigned to the label, before the END directive is executed.

Symbol Definition

EQU Equate symbol value

Format: label EQU exp

The EQU directive assigns the value of the expression in the operand field to the symbol in the label field. The value assigned is permanent, so the label may not be defined anywhere else in the program. The expression must not contain forward references.

EQUR Equate register value

Format: label EQUR register

This directive allows the user to equate one of the processor registers with a user symbol. Only the Address and Data registers are valid, so special symbols like SR, CCR and USP are illegal here. The register assigned is permanent, so the label cannot be defined anywhere else in the program. The register must not be a forward reference to another EQUR statement.

SET Set symbol value

Format: label SET exp

The SET directive assigns the value of the expression in the operand field to the symbol in the label field. SET is identical to EQU, apart from the fact that the assignment is temporary, and can be changed later on in the program. The expression cannot contain forward references, and no forward references are allowed to symbols which are defined using SET.

Data Definition

DC Define Constant

Format:			
	[label]	DC.B	list
	[label]	DC.W	list
	[label]	DC.L	list

The DC directive defines a constant value in memory. It may have any number of operands, separated by commas (","). The values in the list must be capable of being held in the data location whose size is given by the size specifier on the directive. If no size specifier is given, a default of ".W" is assumed. If the size is ".B", then there is one other data type which can be used; that of the ASCII string. This is an arbitrarily long series of ASCII characters, contained within quotation marks. As with ASCII literals, if a quotation mark is required in the string, then two must be entered. If the size is ".W" or ".L", then the assembler aligns the data onto a word boundary.

DS Define Storage

Format:			
	[label]	DS.B	absexp
	[label]	DS.W	absexp
	[label]	DS.L	absexp

The DS directive is used to reserve memory locations, but does not initialise them in any way. The amount of space allocated depends on the data size (given by the size specifier on the directive), and the value of the expression in the operand field. This is interpreted as the number of data items of that size to allocate. As with DC, if the size specifier is ".W" or ".L", the space is aligned onto a word boundary; thus "DS.W 0" will have the effect of aligning to a word boundary only. See CNOP for a more general way of handling alignment. If no size specifier is given, a default of ".W" is assumed.

Listing Control

PAGE Page Throw

Format: PAGE

Unless paging has been inhibited, advance the assembly listing to the top of the next page. The PAGE directive does not appear on the output listing.

LIST Enable Listing

Format: LIST

The LIST directive enables the production of the assembly listing file. Listing continues until either an END or a NOLIST directive is encountered. This directive is only active if a listing file is being produced. The LIST directive does not appear on the output listing.

NOLIST Disable Listing

Format: NOLIST
NOL

The NOLIST directive (for which NOL is a synonym), disables production of the assembly listing file. Listing ceases until either an END or a LIST directive is encountered. The NOLIST directive does not appear on the program listing.

SPC Space Blank Lines

Format: SPC number

The SPC directive outputs the number of blank lines given by the operand field, to the assembly listing. The SPC directive does not appear on the program listing.

NOPAGE Disable Paging

Format: NOPAGE

The NOPAGE directive disables the printing of page throws and title headers on the assembly listing. It is on by default if listing is to a file or device, and off by default if the listing is to the screen.

LLEN Set Line Length

Format: LLEN number

The LLEN directive sets the line length of the assembly listing file, to the value given in the operand field. The value must lie between 60 and 132, and can only be set once in the program. The LLEN directive is not listed. The default is 132.

PLEN Set Page Length

Format: PLEN number

The PLEN directive sets the page length of the assembly listing file, to the value given in the operand field. The value must lie between 24 and 100, and can only be set once in the program. The default is 60.

TTL Set Program Title

Format: TTL title string

The TTL directive sets the title of the program to the string given in the operand field. This string is used as the page heading in the assembly listing. The string starts at the first non-blank character after the TTL, and continues until the end of line. It must not be longer than 40 characters in length. The TTL directive does not appear on the program listing.

NOOBJ Disable Object Code Generation

Format: NOOBJ

The NOOBJ directive disables the production of the object code file at the end of assembly. This directive disables the production of the code file even if a file name was specified when the assembler was started.

FAIL Generate a user error

Format FAIL

The FAIL directive causes the assembler to flag an error for this input line.

Conditional Assembly

CNOP Conditional NOP

Format: [label] CNOP number, number

This directive is an extension from the Motorola standard and allows a section of code to be aligned on any boundary. In particular, it allows any data structure or entry point to be aligned to a long word boundary.

The first expression represents an offset, while the second expression represents the alignment required for the base. The code is aligned to the specified offset from the nearest required alignment boundary. Thus

CNOP 0,4

will align code to the next long word boundary while

CNOP 2,4

will align code to the word boundary 2 bytes beyond the nearest long word aligned boundary.

IFEQ Assemble if Equal
IFNE Assemble if Not Equal

Format: IFEQ absexp
IFNE absexp

The IFEQ and IFNE directives are used to enable or disable assembly, depending on the value of the expression in the operand field. The value is assumed to be EQUAL if it is zero, and NOT EQUAL otherwise. Thus the assembly is disabled if the operand is non zero for IFEQ, or zero for IFNE. The conditional assembly switch remains active until a matching ENDC statement is found. Conditional assembly switches can be nested arbitrarily, and each level of nesting must be terminated by a matching ENDC.

ENDC End conditional assembly

Format: ENDC

The ENDC directive is used to terminate conditional assembly, set up using the IFEQ and IFNE directives. ENDC matches the most recently encountered IFEQ or IFNE.

Macro Directives

MACRO Start a macro definition

Format: label MACRO

This introduces a macro definition, which is terminated by ENDM. The macro is given the name of the label, and subsequent uses of that label as an operand will cause the contents of the macro to be expanded and inserted into the source code. A macro can contain any opcode, most assembler directives or any previously defined macro. When a macro label is used as an operand it may not have a label field on the same line, although an otherwise blank line containing a label may be placed before the line to get the required effect. Code generated by macro expansion is marked with a '+' sign in the listing. When a macro name is used it may be followed by a number of arguments, separated by commas. If the argument is to contain a space (for example, a string containing a space) then the entire argument must be enclosed by '<' (less than) and '>' (greater than) symbols.

The source code entered after a MACRO directive and before an ENDM directive is stored up and saved as the contents of the macro. The code can contain any normal source code; in addition the symbol '\' (backslash) has a special meaning. Backslash followed by a number n indicates that the value of the nth argument is to be inserted into the code. If the nth argument is omitted then nothing is inserted. Backslash followed by the symbol '@' will cause the text '@nnn' to be generated, where nnn is the number of times the \@ combination has been encountered. This is normally used to generate unique labels within a macro.

Macro definitions may not be nested, i.e. a macro cannot be defined within a macro, although a previously defined macro may be called. There is a limit to the level of nesting of macro calls which is currently set at ten.

Macro expansion stops when the end of the stored macro text is encountered, or when an MEXIT directive is found (see below).

ENDM Terminate a macro definition

Format: ENDM

This terminates a macro definition introduced by a MACRO directive.

MEXIT Exit from macro expansion

Format: MEXIT

This directive is used to exit from macro expansion mode. It is normally used in conjunction with the IFEQ and IFNE directives, and allows conditional expansion of macros. Once the directive is executed, the assembler stops expanding the current macro as though there was no more stored text to include.

External Symbols

XDEF Define an internal label as an external entry point

Format: XDEF label [,label]

One or more labels may follow the XDEF directive; they may be absolute or relocatable but the name must be less than 8 characters. Each label defined here will generate an external symbol definition. References to the symbol can be made in other modules (possibly from a high level language) and the references satisfied by a linker. If this directive or the next is used then the code produced by the assembler is not directly executable.

XREF Define an external name

Format: XREF label[,label]

One or more labels which must not have been defined elsewhere in the program follow the XREF directive. Subsequent uses of the label must be at locations aligned to a word boundary and will cause an external reference to be generated for that label. The label will be used as if it referred to an absolute value, and the actual value used will be filled in from another module by the linker. The value may only be used where a 32 bit or 16 bit value would be valid. The linker will also generate any relocation information that may be required in order for the resulting code to be relocatable.

External symbols are normally used as follows. A routine in one program segment is specified as an external definition by placing a label at the start of the routine and quoting the label after an XDEF directive. Another program may call that routine by declaring a label via the XREF directive and then jumping to the label so declared. Data areas may be accessed in the same way so long as the values are either 32 or 16 bits.

General Directives

GET Insert an external file

Format: GET "file name"

The GET directive allows the inclusion of external files into the program source. The file which is inserted is given by the string descriptor in the operand field. GET directives can be nested to a depth of three. The file name must be enclosed in quotes as shown. This is especially useful when a standard set of macro definitions or EQUs are required in several programs; the definitions are placed in a single file and other programs reference them by means of a suitable GET. It is often convenient to place NOLIST and LIST directives at the head and tail of files intended to be included via GET.

2.7 Example programs

Example Program 1

A simple program to read and write a string.

```
*
* QDOS Request codes
*
MT_FRJOB    EQU     $05
IO_OPEN     EQU     $01
IO_CLOSE    EQU     $02
IO_FLINE    EQU     $02
IO_SSTRG    EQU     $07
*
* Macros
*
QDOS        MACRO
            MOVEQ   #\1,D0
            TRAP    #\2
            ENDM
*
TIDYUP      MACRO
            QDOS    IO_CLOSE,2  Close channel
            MOVEQ   #-1,D1
            MOVEQ   #0,D3       Cancel this job
            QDOS    MT_FRJOB,1
*
* In case CALLed
            MOVEQ   #0,D0       Return code - all OK
            RTS
            ENDM
*
* Main program
*
* Open stream
*
            MOVEQ   #-1,D1      Current Job
            MOVEQ   #2,D3       Exclusive Device
            LEA.L   DEVNAME,A0  Pointer to device name
            QDOS    IO_OPEN,2   Open stream
*
* Print prompt
*
            MOVEQ   #18,D2      Length of string
            MOVEQ   #-1,D3      Infinite timeout
            LEA.L   PROMPT,A1   Pointer to string
            QDOS    IO_SSTRG,3  Print prompt
*
* Read reply
*
            LEA.L   BUFFER,A3   Pointer to buffer
            LEA.L   2(A3),A1    Skip first word
            MOVEQ   #30,D2      Length of buffer
            QDOS    IO_FLINE,3  Read input,D1 gets set
*                               to nbytes read
            MOVE.W  D1,(A3)     Save no. of bytes read
*
* Print message
*
            MOVEQ   #6,D2       Length of text
            MOVEQ   #-1,D3      Infinite Timeout
            LEA.L   MESS,A1     Pointer to message
            QDOS    IO_SSTRG,3  print message
            MOVEQ   #0,D2       Clear D2
            MOVE.W  (A3),D2     Length of name
            LEA.L   2(A3),A1    Pointer to name
            QDOS    IO_SSTRG,3  Print name
*
* Tidy up
*
            TIDYUP
*
* Data Section
*
BUFFER      DS.W    1
            DS.B    30
DEVNAME     DC.W    4
            DC.B    'CON_'
MESS        DC.B    'Hello '
PROMPT      DC.B    'Enter your name : '
*
            END
```

Example Program 2

A routine to examine the contents of a part of memory.

```
*
MT_FRJOB    EQU     $05
IO_OPEN     EQU     $01
IO_CLOSE    EQU     $02
IO_SBYTE    EQU     $05
IO_SSTRG    EQU     $07
NADDR       EQU     100
*
* Macro
*
QDOS        MACRO
            MOVEQ   #\1,D0
            TRAP    #\2
            ENDM
*
TIDYUP      MACRO
            QDOS    IO_CLOSE,2  Close channel
            MOVEQ   #-1,D1
            MOVEQ   #0,D3       Cancel this job
            QDOS    MT_FRJOB,1
*
* In case CALLed
            MOVEQ   #0,D0
            RTS
            ENDM
*
* Main program
*
* Open stream
            MOVEQ   #-1,D1      Current Job
            MOVEQ   #2,D3       Exclusive Device
            LEA.L   DEVNAME,A0  Pointer to device name
            QDOS    IO_OPEN,2   Open stream
*
            LEA.L   $28000,A4   Start address
            MOVEQ   #(NADDR-1),D4  (No. of addresses
*                               to be examined -1)
*
* Main Loop
*
LOOP
*
* Print Address as 8 hex digits
            MOVEQ   #10,D2      Length of string
            MOVEQ   #-1,D3      Infinite timeout
            LEA.L   MESS1,A1    Pointer to string
            QDOS    IO_SSTRG,3  Print string
```

```
*
            MOVE.L   A4,D1          Address into D1
            BSR      WRITE32BITS    Print address
*
* Print contents of address
            MOVEQ    #14,D2         Length of text
            MOVEQ    #-1,D3         Infinite Timeout
            LEA.L    MESS2,A1       Pointer to string
            QDOS     IO_SSTRG,3     Print string
*
            MOVE.L   (A4)+,D1       Contents to D1 & inc
*                                   pointer
            BSR      WRITE32BITS    Print contents
*
* Output L/F
            MOVEQ    #$0A,D1        L/F into D1
            BSR      WRITECHAR      Write the character
*
* All addresses examined? Loop if not
            DBRA     D4,LOOP
* Tidy up
*
            TIDYUP
*
WRITE32BITS
            SWAP     D1
            BSR.S    WRITE16BITS      Write top 16 bits
            SWAP     D1               and drop through to
*                                    write out the low
*                                    16 bits
WRITE16BITS
            ROR.W    #8,D1
            BSR.S    WRITE8BITS         Write top 8 bits
            ROL.W    #8,D1              (of 16) and drop
*                                     through to write out
*                                     the low 8 bits
WRITE8BITS
            ROR.B    #4,D1
            BSR.S    WRITE4BITS         Write top 4 bits
            ROL.B    #4,D1              (of 8) and drop
*                                     through to write
*                                     out the low 4 bits
WRITE4BITS
            MOVE.L   D1,-(SP)           Save D1 on stack
            ANDI.B   #$0F,D1            Mask to retain
*                                     low 4 bits
            ADDI.B   #'0',D1            Add character
*                                     code for zero
            CMPI.B   #'9',D1            Is it > charcter
*                                     nine
            BLS.S    WRITE4BITS1        No, so write
*                                     character out
            ADDI.B   #'A'-'9'-1,D1      Yes, so convert
*                                     to range A-F
WRITE4BITS1
            BSR.S    WRITECHAR
            MOVE.L   (SP)+,D1           Restore D1
            RTS
WRITECHAR
            MOVEM.L  D0/D3/A1,-(SP)     Stack registers
*                                     used/corrupted
            MOVE.W   #-1,D3             Infinite Timeout
            MOVEQ    #IO_SBYTE,D0       Code to send
*                                     1 byte
            TRAP     #3                 Byte to be sent
*                                     in D1
            MOVEM.L  (SP)+,D0/D3/A1     Restore
*                                     registers
            RTS
*
* Data Section
*
DEVNAME     DC.W     4
            DC.B     'CON '
MESS1       DC.B     'Address : '
MESS2       DC.B     '   Contents : '
            END
```

Example Program 3

A program to pan a string inside a window which can be increased/decreased in size as well as moved around the screen.

```
* Constant section
*
            TTL   Demonstration - Constants
*
* QDOS requests
*
* Trap #1 codes
*
MT_FRJOB    EQU   5
* Trap #2 codes
IO_OPEN     EQU   1
IO_CLOSE    EQU   2
* Trap #3 codes
IO_FBYTE    EQU   1
IO_SBYTE    EQU   5
SD_BORDR    EQU  $C
SD_WDEF     EQU  $D
SD_CURE     EQU  $E
SD_PIXP     EQU $17
SD_PAN      EQU $18
SD_CLEAR    EQU $20
SD_SETIN    EQU $29
SD_SETSZ    EQU $2D
*
* Colours
*
C_GREEN     EQU   4
C_WHITE     EQU   7
C_CHAR1     EQU   C_GREEN
C_CHAR2     EQU   C_WHITE
*
* Key Codes
*
K_ENTER     EQU   $0A
K_SPACE     EQU   $20
K_UP        EQU   $D0
K_DOWN      EQU   $D8
K_LEFT      EQU   $C0
K_RIGHT     EQU   $C8
K_ALT_UP    EQU   $D1
K_ALT_DOWN  EQU   $D9
K_ALT_LEFT  EQU   $C1
K_ALT_RIGHT EQU   $C9
*
* A few useful constants
*
CHWIDTH     EQU   16
CHHEIGHT    EQU   20
MINH        EQU   CHHEIGHT+2
MINW        EQU   CHWIDTH*6+4
MAXX        EQU   512-CHWIDTH-4
MAXY        EQU   256-CHHEIGHT-2
BWIDTH      EQU   1             Width of window border
WIDTH       EQU   MINW
HEIGHT      EQU   MINH
X           EQU   304
Y           EQU   200
*
* Names for registers
*
WW          EQUR  D4        Current window width
WH          EQUR  D5        .. and height
WX          EQUR  D6        X position of top left
*                          corner
WY          EQUR  D7        .. and Y position
*
            PAGE
            TTL   Demonstration - Macros
            SPC   2
* Macros
            SPC   2
*
* A generalised call for cursor positioning, scrolling,
* panning etc. Timeout is infinite and A1 is assumed to
* contain channel to screen.
*
SCROPS      MACRO
            MOVEQ   #\1,D1
            MOVEQ   #\2,D2
            MOVEQ   #\3,D0          Opcode into D0
            MOVEQ   #-1,D3          Timeout
            LEA.L   WINDOW,A1       Address of window block
            TRAP    #3              Do it
            ENDM
            SPC   2
*
* A generalised call for QDOS requests
*
QDOS        MACRO
            MOVEQ   #\1,D0          Trap code
            TRAP    #\2             Do it
            ENDM
*
TIDYUP      MACRO
            QDOS    IO_CLOSE,2      Close console channel
            MOVEQ   #-1,D1
            MOVEQ   #0,D3           Cancel this job
            QDOS    MT_FRJOB,1      Terminate this job
*
* In case CALLed
            MOVEQ   #0,D0           Return code - all OK
            RTS
            ENDM
*
            PAGE
            TTL   Demonstration - Main program
*
* Main Program
*
```

```
INIT
*
* Open Console stream
*
        MOVEQ   #-1,D1          Current Job
        MOVEQ   #2,D3           Exclusive device
        LEA.L   DEVNAME,A0      Pointer to device name
        QDOS    IO_OPEN,2       Open channel, ID in A0
*
* Set default window
*
        SCROPS  3,1,SD_SETSZ    Large chars
        SCROPS  0,0,SD_CURE     Enable cursor
*
* Initialise window section
*
        MOVE.W  (A1)+,WW        Width
        MOVE.W  (A1)+,WH        Height
        MOVE.W  (A1)+,WX        X
        MOVE.W  (A1)+,WY        Y
        LEA.L   LOGO,A4         Use A4 to point to
*                               current character
        LEA.L   LOGOE,A5        And A5 to hold end
        BRA     SETWIN9         Display window and enter
*                               loop
*
* Main loop
*
SETWINDOW
        BSR     VDU_RDCH        Get character or timeout
        BEQ.S   SETWIN0         Character ready, so
*                               handle that case
*
* No character typed yet, so handle scrolling of name
*
        SCROPS  -CHWIDTH,0,SD_PAN
        MOVE.B  10(A4),D1       Get colour for this char
        QDOS    SD_SETIN,3      Change colour
        MOVE.B  (A4)+,D1        Get char
        QDOS    IO_SBYTE,3      Print character (A0 has
*                               channel ID)
        BSR     SETCURSOR       Set the cursor back
        CMPA.L  A4,A5           Reached end?
        BNE.S   SETWINDOW       No, wonderful
        LEA.L   LOGO,A4         Yes, reset ptr
*
* Handle window movement
*
SETWIN0
        CMPI.B  #K_ENTER,D1     Was it ENTER ?
        BEQ     FINISH          End program
        CMPI.B  #K_SPACE,D1     Was it SPACE ?
        BEQ     SETWIN9         Yes, so redraw window
SETWIN1
        CMPI.B  #K_UP,D1        Up arrow?
        BNE.S   SETWIN2
        CMPI.W  #CHHEIGHT,WY    Check for y <= CHHEIGHT
        BLT.S   SETWINDOW       If so, don't alter it -
*                               Wait for next char
        SUBI.W  #CHHEIGHT,WY    Otherwise take CHHEIGHT
*                               away
        BRA     SETWIN9         Redraw window
SETWIN2
        CMPI.B  #K_DOWN,D1      Down arrow ?
        BNE.S   SETWIN3
        MOVE.W  WH,D3           D3 = Height
        ADD.W   WY,D3           Add Y to it
        CMPI.W  #MAXY,D3        Is Y + Height >= MAXY
        BGE     SETWINDOW       Yes - Wait for next char
        ADDI.W  #CHHEIGHT,WY    No so add CHHEIGHT to Y
        BRA     SETWIN9         Redraw window
SETWIN3
        CMPI.B  #K_LEFT,D1      Left arrow ?
        BNE.S   SETWIN4
        CMPI.L  #CHWIDTH,WX     Is X < chsize
        BLT     SETWINDOW       Yes, so ignore
        SUBI.W  #CHWIDTH,WX     X=X-chsize
        BRA.S   SETWIN9         Redraw window
SETWIN4
        CMPI.B  #K_RIGHT,D1     Right arrow ?
        BNE.S   SETWIN5
        MOVE.W  WW,D3           Width
        ADD.W   WX,D3           plus X
        CMPI.W  #MAXX,D3        Are we at end of screen
        BGE     SETWINDOW       Yes, so ignore
        ADD.W   #CHWIDTH,WX     X=X+chsize
        BRA.S   SETWIN9         Redraw window
SETWIN5
        CMPI.B  #K_ALT_UP,D1    ALT Up arrow ?
        BNE.S   SETWIN6
        CMPI.W  #MINH,WH        Is Height<=MINH
        BLE     SETWINDOW       Yes, so ignore
        SUB.W   #CHHEIGHT,WH    Height=Height-CHHEIGHT
        BRA.S   SETWIN9         Redraw window
SETWIN6
        CMPI.B  #K_ALT_DOWN,D1  ALT Down arrow ?
        BNE.S   SETWIN7
        MOVE.W  WH,D3           Height
        ADD.W   WY,D3           Add Y
        CMPI.W  #MAXY,D3        Is Height+Y >= MAXY
        BGE     SETWINDOW       Yes, so ignore
        ADD.W   #CHHEIGHT,WH    Height=Height+CHHEIGHT
        BRA.S   SETWIN9
SETWIN7
        CMPI.B  #K_ALT_LEFT,D1  ALT Left Arrow ?
        BNE.S   SETWIN8
        MOVE.W  WW,D3           Width
        SUB.W   #CHWIDTH,D3     Minus CHWIDTH
        CMPI.W  #MINW,D3        Is it <= MINW
        BLT     SETWINDOW       Yes, so ignore
        SUB.W   #CHWIDTH,WW     Reduce width
        BRA.S   SETWIN9         Redraw window
SETWIN8
        CMPI.B  #K_ALT_RIGHT,D1 ALT Right arrow ?
        BNE     SETWINDOW
        MOVE.W  WW,D3           Width
        ADD.W   WX,D3           Plus X
        CMPI.W  #MAXX,D3        Is it >= MAXX
        BGE     SETWINDOW       Yes, so ignore
        ADD.W   #CHWIDTH,WW     Increase Width
SETWIN9
        SCROPS  0,1,SD_BORDR    Remove old border
        SCROPS  0,0,SD_CLEAR    And clear old window
        LEA.L   WINDOW,A1       Get address of buffer
        MOVE.W  WW,(A1)+        Restore Width
        MOVE.W  WH,(A1)+        Restore Height
        MOVE.W  WX,(A1)+        Restore X
        MOVE.W  WY,(A1)         Restore Y
        SCROPS  C_WHITE,BWIDTH,SD_WDEF    Redefine
*                               window
        SCROPS  0,0,SD_CLEAR    Clear Screen
        BSR.S   SETCURSOR       Set the cursor correct
        LEA.L   LOGO,A4         Reset character pointer
        BRA     SETWINDOW       Go back and get next ke
*
* Tidy up
*
FINISH
        TIDYUP
*
* Position cursor to standard place (PIXP specified
* relative to window)
*
SETCURSOR
        MOVEQ   #0,D2           Cursor at top position
        MOVE.W  WW,D1           Right hand edge x-coord
        SUBI.W  #CHWIDTH*2+5,D1 Less two chars width
*                               and allow for border
        MOVEQ   #-1,D3          Timeout
        QDOS    SD_PIXP,3       Position cursor
        RTS
*
* Read one char from the keyboard
*
VDU_RDCH
        MOVEQ   #30,D3          Some timeout
        QDOS    IO_FBYTE,3      Get a character in D1.B
*                               (A0 has channel ID)
        TST.L   D0              Check for timeout =0 if
*                               char typed
        RTS
*
* Data area
*
* Window co-ordinates
*
WINDOW  DC.W    WIDTH
        DC.W    HEIGHT
        DC.W    X
        DC.W    Y
*
DEVNAME DC.W    22
        DC.B    'CON_100x22a304x200_128'
LOGO    DC.B    'MetaComCo '
LOGOE   DC.B    C_CHAR1,C_CHAR2,C_CHAR2,C_CHAR2
        DC.B    C_CHAR1,C_CHAR2,C_CHAR2
        DC.B    C_CHAR1,C_CHAR2,C_CHAR2
*
        END
```

Appendix A: The 68000 instruction set

The address modes are represented as follows:

An	Any address register
Dn	Any data register
Rn	Any register
(An)	Address register indirect
d(An)	with displacement
-(An)	with predecrement
(An)+	with postincrement
<ea>	Any address mode
<aea>	Alterable address mode
<cea>	Control address mode
<dea>	Data address mode
<caea>	Control alterable address mode
<daea>	Data alterable address mode
<maea>	Memory alterable address mode
<rl>	Register list
<imm>	Immediate data

The different categories of effective address are classified in the following table:

Addressing modes	*Data*	*Memory*	*Control*	*Alterable*
Dn	X	-	-	X
An	-	-	-	X
(An)	X	X	X	X
-(An)	X	X	-	X
(An)+	X	X	-	X
d(An)	X	X	X	X
d(Rn,An)	X	X	X	X
Absolute	X	X	X	X
PC relative	X	X	X	-
PC relative index	X	X	X	-
Immediate	X	X	-	-

The sizes for each instruction are given as B for byte, W for word and L for long. Under the column 'Modes' are listed examples of the various forms of address modes. If a number of instructions have the same form, then only the first one is used in the examples. Where the syntax differs for different intructions, all possible variations are included.

Description	*Name*	*Modes*	*Size*
Add binary	ADD	<ea>,Dn	BWL
		Dn,<maea>	
	ADDA	<ea>,An	WL
	ADDI	#<imm>,<daea>	BWL
	ADDQ	#<imm>,<aea>	BWL
	ADDX	Dn,Dn	BWL
		-(An),-(An)	
Add decimal	ABCD	Dn,Dn	B
		-(An),-(An)	
Arithmetic shift left	ASL	Dn,Dn	BWL
		#<imm>,Dn	
		<maea>	
Arithmetic shift right	ASR		BWL
Bit test and change	BCHG	Dn,<daea>	B L
		#<imm>,<daea>	
Bit test and clear	BCLR		B L
Bit test and set	BSET		B L
Bit test	BTST	Dn,<dea>	B L
		#<imm>,<dea>	
Branch on condition	Bcc	<label>	BW
Branch unconditionally	BRA		BW
Branch to subroutine	BSR		BW
Check and possibly TRAP	CHK	<dea>,Dn	W
Compare	CMP	<ea>,Dn	BWL
	CMPA	<ea>,An	WL
	CMPI	#<imm>,<daea>	BWL
	CMPM	(An)+,(An)+	BWL
Compare with zero	TST	<daea>	BWL
Decrement test & branch	DBcc	Dn,<label>	W
Decrement & branch	DBRA		W

Description	*Name*	*Modes*	*Size*
Division - signed	DIVS	<dea>,Dn	W
Division - unsigned	DIVU		W
Exchange registers	EXG	Rn,Rn	L
Jump	JMP	<cea>	-
Jump to subroutine	JSR		-
Load effective address	LEA	<cea>,An	L
Logical AND	AND	<dea>,Dn	BWL
		Dn,<maea>	
	ANDI	#<imm>,<daea>	BWL
	ANDI to CCR	#<imm>,CCR	B
	ANDI to SR	#<imm>,SR	W
Logical exclusive OR	EOR	Dn,<daea>	BWL
	EORI	#<imm>,<daea>	BWL
	EORI to CCR	#<imm>,CCR	B
	EORI to SR	#<imm>,SR	W
Logical OR	OR	<dea>,Dn	BWL
		Dn,<maea>	
	ORI	#<imm>,<daea>	BWL
	ORI to CCR	#<imm>,CCR	B
	ORI to SR	#<imm>,SR	W
Logical shift left	LSL	Dn,Dn	BWL
		#<imm>,Dn	
		<maea>	
Logical shift right	LSR		BWL

Description	Name	Modes	Size
Move data	MOVE	<ea>,<daea>	BWL
	MOVEA	<ea>,An	WL
Move multiple	MOVEM	<rl>,-(An)	WL
		<rl>,<caea>	
		(An)+,<rl>	
		<cea>,<rl>	
Move to peripheral	MOVEP	Dn,d(An)	WL
		d(An),Dn	
	MOVEQ	#<imm>,Dn	L
	MOVE to CCR	<dea>,CCR	W
	MOVE to SR	<dea>,SR	W
	MOVE from SR	SR,<daea>	W
	MOVE USP	USP,An	L
		An,USP	
Multiply - signed	MULS	<dea>,Dn	W
Multiply - unsigned	MULU		W
Negate binary	NEG	<daea>	BWL
	NEGX		BWL
Negate decimal	NBCD	<daea>	B
No operation	NOP		
Logical complement	NOT	<daea>	BWL
Push effect address	PEA	<cea>	L
Reset	RESET		-
Return from exception	RTE		-
Return and restore CCR	RTR		-
Return from subroutine	RTS		-

Description	Name	Modes	Size
Rotate left	ROL	Dn,Dn	BWL
		#<imm>,Dn	
		<maea>	
	ROXL		BWL
Rotate right	ROR		BWL
	ROXR		BWL
Set from condition	Scc	<daea>	B
Set to zero	CLR	<daea>,Dn	BWL
Sign extend	EXT	Dn	WL
Stop execution & wait	STOP	#<imm>	-
Subtract binary	SUB	<ea>,Dn	BWL
		Dn,<maea>	
	SUBA	<ea>,An	WL
	SUBI	#<imm>,<daea>	BWL
	SUBQ	#<imm>,<aea>	BWL
	SUBX	Dn,Dn	BWL
		-(An),-(An)	
Subtract decimal	SBCD	Dn,Dn	B
		-(An),-(An)	
Subroutine link	LINK	An,#<imm>	-
Subroutine unlink	UNLK	An	-
Swap register halves	SWAP	Dn	W
Test bit and set	TAS	<daea>	B
Trap exception	TRAP	#<imm>	-
TRAP if overflow	TRAPV		-

Conditional tests

The following may be specified as the conditional test in the branch to condition (Bcc), decrement test and branch (DBcc) and set from condition (Scc) instructions. Also T and F may be used to indicate True and False in DBcc and Scc instructions.

In the following table, C indicates that the C status bit must be set, and C' means that the bit must be unset for the condition to be true. Two symbols, & and |, are used to connect conditions. & means that both must be true; | means that either may be true.

Condition	Name	Test
Carry clear	CC	C'
Carry set	CS	C
Equal	EQ	Z
Not equal	NE	Z'
Plus	PL	N'
Minus	MI	N
Overflow clear	VC	V'
Overflow set	VS	V
High	HI	C' & Z'
Low or same	LS	C | Z
High or same	HS	C'
Low	LO	C
Greater than	GT	(N & V & Z')|(N' & V' & Z')
Greater than or equal	GE	(N & V)|(N' & V')
Less than or equal	LE	Z|(N & V')|(N' & V)
Less than	LT	(N & V')|(N' & V)

Appendix B: Installation

Changing the default window

Both the editor and the assembler allow the window which is to be used to be altered as part of the initialisation sequence. If this option is not required then the default window is used. This is initially the same as the window used during the start of the program, but if required the default window may be altered permanently by patching the programs. This is useful where a certain window size and position is always required and means that the window does not have to be positioned correctly each time the program is run.

Changing the default drive name

For those users who upgrade their QLs with disc drives, there is the possibility of changing the default drive to something other than mdv1. This option will not be given when installing the editor ED since it can be EXECed from any device.

The INSTALL program

The program INSTALL is supplied on the distribution microdrive to perform both of the above tasks. It is run by the command

```
LRUN mdv1_install
```

The program starts by asking whether the default window is to be set up for TV or monitor mode. The minimum window size is greater in TV mode because the characters used are larger. You should answer T if you are setting the default for use with TV mode and M if you are setting it for use with monitor mode. Note that the current mode in use is of no consequence.

The standard window will appear on the screen and can be moved by means of the cursor keys and altered in size by means of ALT cursor keys. This is similar to the mechanism used when altering the window during normal program initialisation. Once the window is in the right place and of the desired size, press ENTER.

The program now asks for the name of the file which is to be modified. If you wished to alter the editor then the file would probably be something like 'mdv1_ed'. The next item requested is the name of the program. When a new job such as the editor or the assembler is running on the QL, it has a name associated with it. This can be inspected by suitable utilities. The name is six characters long, and whatever is typed here is used as the name and forced to the correct length. The name is of little importance except for job identification.

In the case of the assembler the program will then go on to ask for a default drive name where it should look for its overlays. If you do not wish to change the default drive name the reply should be

```
MDV1
```

(Note - the reply must not be MDV1_). If you do wish to change the default drive name the reply should be the device name, for example

FLP1

In this latter case the assembler will append 'FLP1_' to its overlays before attempting to load them.

The INSTALL program will then modify the file specified. INSTALL can be run as many times as you like to alter the default window of the editor or the assembler. It is unlikely to be useful with programs other than those distributed by Metacomco that provide user selection of an initial window such as Metacomco's Assembler, LISP and BCPL.

Appendix C: The Linker Program

The commands XDEF and XREF make it possible for an assembler program to refer to labels not defined within the program itself, but defined elsewhere in another quite separate assembler program.

When a program containing XDEFs and/or XREFs is assembled, the output module produced is in a special format (Sinclair Relocatable Binary Object File Format). This file cannot be directly EXECed but must be linked with the module(s) in which the external labels are defined. The linker program supplied on the microdrive will link such modules together to produce an EXECable file.

Note that the linker is only capable of linking two or more modules produced by the Metacomco assembler. It is not suitable for linking Assembler modules to modules produced by compiling a high level language program.

To run the linker one types

```
EXEC MDV1_LINKER
```

or

```
EXEC_W MDV1_LINKER
```

The linker first asks for a workspace size. The reply given should be a number (of bytes) optionally followed by a 'K' which converts the requested number to Kilobytes. The minimum size which can be asked for is 1K. If no number is typed the linker will use its default workspace size (10K).

It will then go on to ask for each of the Binary files (up to a maximum of 20 code files produced by the assembler) in turn and, having read each file, will report the file's length. This length does not include all the extra bytes put in the file by the assembler to create the special file format. When just ENTER is typed in reply to the request for a binary file the linker will start its second pass which entails re-reading each file in turn.

After processing all the files, the linker will ask for a name for an output file and then a stack size (in bytes) which will be used with the program when it is run.

An example containing external references now follows. The program has been split into four separate programs which must be assembled separately and the resulting code files linked together.

Programa

```
* Program a

MT_FRJOB        EQU     $05
IO_OPEN         EQU     $01
IO_CLOSE        EQU     $02

QDOS    MACRO
        MOVEQ   #\1,D0
        TRAP    #\2
        ENDM

        XREF    PRINTP,PRINTM

* Open stream

OPENFL  MOVEQ   #-1,D1          Current job
        MOVEQ   #2,D3           Exclusive device
        LEA.L   DEVNAME,A0      Point to device name
        QDOS    IO_OPEN,2

* Print prompt

        JSR     PRINTP

* Print message

        JSR     PRINTM
* Tidy up
        QDOS    IO_CLOSE,2
        MOVEQ   #-1,D1
        MOVEQ   #0,D3           Cancel this job
        QDOS    MT_FRJOB,1
* In case called
        MOVEQ   #0,D0           Return code - all OK
        RTS                     And home

* Data Section

DEVNAME DC.W    4
        DC.B    'CON_'

        END
```

Programb

```
* Program b

IO_SSTRG        EQU     $07

QDOS    MACRO
        MOVEQ   #\1,D0
        TRAP    #\2
        ENDM

        XREF    READREP,MESS,PROMPT,BUFFER
        XDEF    PRINTP,PRINTM

* Print prompt

PRINTP
        MOVEQ   #18,D2          Length of text
        MOVEQ   #-1,D3          Timeout
        LEA.L   PROMPT,A1       Message
        QDOS    IO_SSTRG,3
* Read reply
        JSR     READREP         Get reply
        RTS                     Return to calling
*                               program

* Print message

PRINTM
        MOVEQ   #6,D2           Length of text
        MOVEQ   #-1,D3          Timeout
        LEA.L   MESS,A1         Message
        QDOS    IO_SSTRG,3
        MOVEQ   #0,D2           Clear D2
        LEA.L   BUFFER,A3       Pointer to name
        MOVE.W  (A3),D2         Length of name
        LEA.L   2(A3),A1        Ptr to name
        QDOS    IO_SSTRG,3
        RTS
*
        END
```

Programc

```
* Program c

IO_FLINE        EQU     $02

        XREF    BUFFER
        XDEF    READREP

* Read reply

READREP
        LEA.L   BUFFER,A3       Ptr to buffer
        LEA.L   2(A3),A1        Skip first word
        MOVEQ   #30,D2          Length of buffer
        MOVEQ   #IO_FLINE,D0
        TRAP    #3
        MOVE.W  D1,(A3)         Save number
*                               of bytes read
        RTS
*
        END
```

Programd

```
* Program d

        XDEF    BUFFER,MESS,PROMPT

* Data Section

BUFFER  DS.W    1
        DS.B    30
MESS    DC.B    'Hello '
PROMPT    .B    'Enter your name : '
*
```