e68k Online Guide html3.2 ©Miguel Filgueirasindex.html, Universidade do Porto, 1992–1997 4 3 1

A Portable Environment for Programming in MC68000 Assembly

Miguel Filgueiras

LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150 Porto, Portugal
`email: mig@ncc.up.pt`

February 92; revised January 1997

1 Introduction

The need for providing a uniform environment for students that start programming in an assembly language and that may use computers with different processors led to the design and implementation of **e68k**, the system described in the sequel. The fact that in previous years the computers available to students of our department had a Motorola MC68000 processor and the obvious benefits in using existing lecture notes on its assembly language ([**?**]) explain the choice of language. Note that the sources of **e68k**e68k.tgz are public and that they may well be used as a skeleton for building a similar environment for another (not too different) language.

2 Using e68k

The main idea behind **e68k** is to provide the user with a programming environment similar to the ones that are traditional for languages like BASIC. Within the environment the user may issue commands to assemble a program, list it, run it (in normal- or debug-mode), clear the memory, and so on.

One difference to traditional interpreter systems is that there is no direct input of the program text from the keyboard. Instead a file containing the text must be specified and it may be altered by calling an editor (without leaving **e68k**). This is the reason for **e68k** asking at the very start for the name of the file that is to be considered initially. The user is free to work with other files, although at any time only one file can be open.

2.1 Some Commands

A brief description of the more frequently used commands will be now given. Note that the set of **e68k** commands may be easily altered. The help facility of **e68k** gives a means of knowing exactly what commands are available in a particular implementation.

Getting help The `help` command, which may be used whenever a command is expected, gives the user information on which commands are available at the moment.

Assembling The `assemble` command opens the file whose name was last given and assembles the program in it. There will be error messages whenever the file cannot be opened or there are syntactic errors in the text. This command can only be used when there is no assembled program in the memory.

Listing The `list` command produces a listing of the program that has been successfully assembled, giving an error if there is none. For each line **e68k** shows the line number, the memory address at which the data or code is located in hexadecimal, and the actual line in the program text. An address will be followed either by a *p* (for *program-segment*), or by a *v* (for *values*, or *data-segment*) — more details on this below.

Running There are two different modes for running a program. The *normal* mode, in which the user has no control on what the program is doing, is entered by using the `run` command. The *debug* mode, which is explained below, gives the user the possibility of following in detail the program execution. It is entered through the use of `debug`, and it has a distinct set of commands (that is why the prompt is different in this mode). Whatever the mode is, at the end of the execution **e68k** issues a message reporting why the program stopped.

Editing The `edit` command allows the user to make changes in the current file by calling a text editor. Which editor is used by default is implementation dependent (for instance, in a UNIX system the shell environment will be searched for a variable EDITOR, which if defined will be taken to specify the editor). The `setedit` command may be used to inform **e68k** of the name of the editor to be called. Whenever possible **e68k** warns the user when listing or debugging a program for a file that has been changed after the assembling of the program.

Using a different file When a program that has been successfully assembled is no longer needed the `clear` command will make **e68k** forget it. To specify that a different file is to be treated in subsequent assembling, listing, or editing operations the `file` command is used. It is an error to issue this command while there is an assembled program in memory.

Quitting The command `quit` terminates the execution of **e68k** and returns control to the operating system

A typical sequence of commands for preparing and running a new program will be: `edit`, to write the program text into the current file, `assemble`, to assemble it, and `run` to execute it if there were no errors during the assembling (otherwise the errors should be corrected by calling again the editor). When the results are not the expected ones, running the program in debug-mode will certainly be helpful in locating the problem.

2.2 Debugging

The **e68k** debugger is modelled after the `dbx` symbolic debugger normally supplied with UNIX systems, its capabilities being, of course, only a small subset of those of `dbx`. The debugger accepts a set of commands that enables the user to execute the program in memory in a controlled way and also to inspect and alter the (simulated) MC68000 registers and memory contents. The help facility may be used to ascertain what commands are available — note that these will change when the program has been run to completion as some of them would not make sense at that time.

When the debugger expects a command it will print the program line containing the instruction that is about to be executed (or, when the program has terminated, that has just been executed) and a special prompt asking for a command. The most important debugger commands are explained in the following description.

Step-by-step execution The `step` command will make **e68k** to execute the assembler instruction in the current line.

Turning the debugger off To run the rest of program in normal-mode, without user intervention, the `nodebug` command is used, while `abort` will cause an immediate exit from the debugger.

Using breakpoints The user may set and delete breakpoints at program lines containing executable instructions. The commands `break` and `delete` are provided for that, breakpoints being referred to by line numbers. The command `cont` continues the program execution until either a breakpoint is reached or the program exits.

Accessing registers The command `regs` prints the registers contents (in hexadecimal), whereas `setreg` alters the contents of a given register. In the latter case, the user can specify an attribute (`.B`, `.W`, or `.L` — the default) after the register name. The value to which the register will be set must be written according to the syntax of the assembler for numeric constants.

Accessing memory The command `mem` prints the contents of a sequence of bytes in memory, while `setmem` alters the contents of such a sequence. With the former the user is asked for an address (in the data-segment) and the length in bytes of the sequence to be displayed. Memory contents will be displayed in hexadecimal and as characters, the character '?' meaning either itself or any non-printable character. With the latter command the user is also asked for an address (also in the data-segment) and for a sequence of values to be loaded into memory. Both the addresses and the values must be given according to the syntax of the assembler for numeric constants.

Listings The `list` command works as described before, while `line` prints the current line.

Restarting The `restart` command forces the execution of the program from the beginning.

3 The Assembly Language

The assembly language accepted by **e68k** is similar to the language described in [?] although the following restrictions and differences are to be taken into account.

Names and expressions Mnemonics, attributes and register names must be in capital letters. Names for labels may have lower-case letters, digits and '_' (the underline character), but must not have periods and must begin with a letter (in either upper- or lower-case). Labels must begin at the first column of the line, and semi-colons are not to be used after them. For the time being, expressions are not allowed.

Comments A line starting with an asterisk, preceded or not by blanks is taken to be a comment and discarded. Any character after a semi-colon in a line will also be discarded.

Attributes Whenever an instruction either has only one possible attribute, or may have different attributes that can be determined by the assembler, **e68k** treats the instruction as unsized and does not allow the user to specify an attribute.

Unimplemented instructions A very few MC68000 instructions were not implemented, in particular those that can only be used in MC68000 supervisor mode. A suitable error message is issued when such an instruction is found.

Directives and pseudo-instructions The following are provided:

- `OPNF` opens a file and returns a file number that will be used for accessing it. It has two arguments: an address register, which must contain the address of a sequence of bytes terminated with a null-byte that specifies the name (or path) of the file, and a data register, whose byte 0 should be 0 for reading, 1 for writing, or 2 for appending (or creating) the file. In the end, the byte 0 of the destination register is set either to the number associated with the file, or to -1 whenever an error occurred (wrong initial value in the data register, too many files open, or the file could not be opened).
- `CLSF` closes the file whose number is given in byte 0 of the data register used as its argument.
- `GETC` may be used either with a single argument or with two arguments. The arguments must be data registers. With a single argument a byte is read from the standard input (normally the keyboard) into the byte 0 of the register. With two arguments, a byte is read from the file whose number (see `OPNF`) is given in byte 0 of the source register into the byte 0 of the destination register. In this case and if the file number is invalid the value -1 is written into the byte 0 of the destination register.

- `PUTC` has either a single argument or two arguments, which must be data register(s). With a single argument this directive puts the contents of the byte 0 of the register into the standard output (normally the screen). With two arguments, byte 0 of the source register specifies the file number (see `OPNF`) where byte 0 of the destination register is to be put. In this case, byte 0 of the destination register will be set to -1 when the given file number is invalid.
- `EXIT` has no arguments and causes the termination of the program being executed.
- `END` is just used to mark the end of the program text. Any line after it will be disregarded.
- `DS` (for *data structure*), may have the usual attributes for byte, word (the default), and long, and has a single argument which is a numeric constant specifying the number of positions of the given length to be allocated in the data-segment.
- `DC` (for *data constants*), may also have the usual attributes as for `DS`, and its argument is a constant list consisting of constants separated by commas. The constants will be put in successive positions of the given length in the data-segment. When the attribute is `.B` a sequence of characters enclosed by double quotes is taken to be a character string, and the ASCII codes of the characters in it are treated as the constants to be stored.

Although not being the case with most computers having a MC68000 processor, **e68k** makes a separation between two memory segments: the *program-segment* and the *data-segment*. This is due to the fact that the internal representation of MC68000 instructions in **e68k** is not at all their binary codes. It would therefore be a bad idea to give the user access to memory locations whose contents are different from the expected ones. When assembling a program **e68k** automatically keeps track of what is to be put in each of both segments, even when executable instructions are mixed with data storage directives. As it is usual with this kind of architecture, the assembly program cannot access the program-segment for storing or getting data and cannot execute instruction codes stored in the data-segment.

The usual big-endian behaviour of the MC68000 is implemented, irrespective of the processor running **e68k**, if **e68k** is compiled with the correct options (as described in the next section).

Another point that is worth mentioning is that messages caused by syntactic errors were intentionally left not too detailed. This may help the user in learning the syntax and in being prepared to use those many hermitic, closed and horrible systems that still users have to use in real life.

4 Implementation

The original implementation of **e68k** was done using the C language in a Sun-4 running UNIX. The dependency with respect to UNIX was deliberately kept low to increase the portability of the environment, and amounts to some routines for checking the modification date of a file, for defining the default editor, and for calling an editor (by executing a shell command of the form `editor filename`).

The **e68k** sources should be compiled with the correct options for the actual processor and operating system. In some Unix environments the options will be set automatically. In any case the header filee68k.h.txt should be checked in what concerns the definitions of `BIGENDIAN`, `LITTLEENDIAN` and `UNIX`, `DOS` identifiers.

The information on the syntax of the assembler instructions and on the commands accepted by **e68k** was as much as possible given, in the program header file, as initial values to arrays. In this way, additions or modifications may be easily done. Also, comments were provided as an explanation on what each routine is expected to do, and whenever some obscure solution was used (fortunately, there are not many of them).

5 Acknowledgements

I would like to express my sincere thanks to Pedro Baltazar Vasconcelos, who gladly accepted to undertake the fastidious task of checking that **e68k** was executing in a honest way the MC68000 instructions. He also started the work on the configuration for different operating systems and processors.

Nelma Moreira and Rogério Reis helped a lot in correcting the bugs that Ana Paula Tomás very effectively detected.

References

[A. P. Tomás 1991] Ana Paula Tomás, Notas e Exercícios de Programação em MC68000. Faculdade de Ciências, Universidade do Porto, 1991.

[Harman and Lawson 1985] Thomas L. Harman and B. Lawson, *The Motorola MC68000 Microprocessor Family*. Prentice-Hall, 1985.