Contents

MOTOROLA
FREEWARE
8-BIT CROSS ASSEMBLERS
USER'S MANUAL
ARCHIMEDES VERSION

v. 3.03

MAURIZIO FERRARI 1992 - 1995
ORIGINALLY EDITED BY
KEVIN ANDERSON
FIELD APPLICATIONS ENGINEER

Support Contact: Maurizio Ferrari, Via Pareto 12, 42020 Albinea (RE), Italy

CHAPTER 1

GENERAL INFORMATION

1.1 INTRODUCTION

This is the user's reference manual for the Archimedes hosted Motorola Freeware 8 bit cross assemblers, ported to the Archimedes to take advantage of the RISC OS facilities, especially throwback (routines by Niklas Röjemo, many thanks for usage permission), drop and drag, and multitasking. It has become much more friendly than the original, command line, PC version. The code is based upon the DECUS C preprocessor and the Motorola freeware assemblers, rewritten in ANSI style, replacing also all the UNIX filing system calls with ANSI ones. It's 99.9 percent compatible with Motorola AS11 source formats, the small difference regarding the way it treats the FCC directive (the original was a possible source of bugs, and buggy too!) and command line syntax. All the RISC OS interface is handled by the appropriate DDEUtils and FrontEnd modules: as they are *not* freeware, I cannot supply them. They are rather common, though, and yes, the Motorola assemblers run from command line too!

The only supported micros are the ubiquitous 68HC11, 68HC05 and 6809. I've been able to thoroughly check the 68HC11 and (partly) the 68HC05. The 6809 code generation relies on the good behaviour of the original program, i.e. any bugs are still here - I've never used a 6809, and I hope I never will...

!asm11/05/09 is freeware. However, as it took many hours to port it, I request the user to send, in exchange, feedback on its use and reports any bugs, so that I may get rid of them, given enough time. You are not allowed to make money reselling it as a software product. If you use it professionally and develop any systems, I'd love to know it. If you think it's valuable, a donation is welcome (say, 5 or 10 £) but *not* requested, as this is freeware.

I couldn't 100% cross check the manual versus the real assembler behaviour, but it should give you enough info to start working. Manual comes in PostScript format, convertible to Draw format thanks to !RiScript, a wonderful utility available from the standard Acorn ftp sites (it's smaller to distribute it this way than the generated Draw files). The more differences you'll report between the manual and !asm11/05/09, the better it will become. Thanks.

You can redistribute this software, according to the following terms:

a) This program is Freeware, and its distribution is bound to the general Motorola Freeware distribution agreement.

b) This program is distributed in its entirety, included copyright messages and these conditions, and in original and unaltered copies. This is only because I want to keep track of it and check any alterations, and be reasonably sure that nobody blames me for someone else's fault. Also, as I said for the manual, bugs will probably be fixed and it's for everybody's good to have it done.

c) You don't flame me for C coding style: I don't like it either, it's difficult to maintain, it was not ANSI, but I'm very grateful to the guy who took the time to write it in the beginning, and to Motorola that provided the original source. I do it in my spare time, my goal was to have it running asap, and it's free! And anyway, from version 3.00 onwards, sources are distributed directly by me, since they don't generally interest the end user. Mail me if you want a copy.

Legally, you are not bound to this agreement, but failing to do so will make your conscience haunt you for the rest of your life!

Neither Motorola, Inc. (obviously, as they don't even know this assembler exist, yet! I will upload it to their BBS anyway) nor Maurizio Ferrari makes any warranty, expressed or implied, with regard to this material including but not limited to merchantability and fitness for a given purpose. The information in this document is subject to change without notice and neither Motorola, Inc. or Maurizio Ferrari assumes any responsibility for any errors which may appear herein. Nor shall Motorola, Inc. or Maurizio Ferrari bear any liability for use of this material with respect compensatory, special, incidental, consequential, or exemplary damages.

The same applies for the public domain DECUS-C preprocessor front-end, I suppose.

This software may contain bugs, and these in turn may cause any disaster you can think of, including but not limited to your wife/girlfriend husband/boyfriend /anyone hitting you hard for staying awake at night debugging nasty code. You have been warned - I know what it's like. Many thanks to my wife Olivia for her patience and my newborn daughter Claudia for sleeping at night...

This manual details the features and capabilities of the cross assemblers, assembler syntax and directives, options, and listings. It is intended as a detailed reference and an introduction for those unfamiliar with Motorola assembler syntax and format.

Assemblers are programs that process assembly language source program statements and translate them into executable machine language object files. A programmer writes his source program using any text editor or word processor that can produce an ASCII text output. With some word processors this is known as "non document" mode. Non document mode produces a file without the non-printable embedded control characters that are used in document formatting. (Caution: assembling a file that has been formatted with embedded control characters may produce assembler errors. The solution is to convert the source file to ASCII text.) Once the source code is written, the source file is assembled by processing the file via the assembler.

Cross assemblers (such as the Motorola Freeware Assemblers) allow source programs written and edited on one computer (the host) to generate executable code for another computer (the target). The executable object file can then be downloaded and run on the target system. In this case the host is a RISC OS computer and the target system is based on a Motorola 8-bit microprocessor (6805, 68HC05, 6809 or 68HC11).

The assemblers are the application directories !asm11 or !asm05 or !asm09, depending on which microprocessor you are writing code for. The details of executing the assembler programs are found in Chapter 3. The assembly language format and syntax for the various processors is very similar with slight variations due to varied programming resources (instructions, addressing modes, and registers). These variations are explained in Appendix B.

Many thanks to my lovely wife Olivia for her support, really!

1.2 ASSEMBLY LANGUAGE

The symbolic language used to code source programs to be processed by the Assembler is called assembly language. The language is a collection of mnemonic symbols representing: operations (i.e., machine instruction mnemonics or directives to the assembler), symbolic names, operators, and special symbols. The assembly language provides mnemonic operation codes for all machine instructions in the instruction set. The instructions are defined and explained in the Programming Reference Manuals for the specific devices, available from Motorola. The assembly language also contains mnemonic directives which specify auxiliary actions to be performed by the Assembler. These directives are not always translated into machine language.

1.3 OPERATING ENVIRONMENT

These assemblers will run on any RISC OS 2 or RISC OS 3 Acorn ARM Computers. The assemblers may be run off of a floppy disk drive or they may be copied onto a hard drive for execution. DDEUtils, FrontEnd and a ThrowBack compliant editor are required for running it under the RISC OS desktop environment.

1.4 C-STYLE PREPROCESSING

The first pass is the C-style preprocessing. It is based on the public domain DECUS C preprocessor, modified to interface to the different (from that of C) assembler syntax. The output files, unless the cpp-only option is selected, is then fed to the assembler. As far as I know, the only incompatible aspect is that a ' or a " in a comment line, unless matched by a closing ' or ", will raise an "unmatched string" error. However, a correctly written asm program should not have unterminated strings after a FCC directive, so it's only a comment problem. Avoid 's and "s in comments or always use them in an even number each line. Should you need a single ' or " in a FCC directive, resort to use it ascii equivalent code.

1.5 ASSEMBLER PROCESSING

The Assembler is a two-pass assembler. During the first pass, the source program is read to develop the symbol table. During the second pass, the object file is created (assembled) with reference to the table developed in pass one. It is during the second pass that the source program listing is also produced.

Each source statement is processed completely before the next source statement is read. As each statement is processed, the Assembler examines the label, operation code, and operand fields. The operation code table is scanned for a match with a known opcode. During the processing of a standard operation code mnemonic, the standard machine code is inserted into the object file. If an Assembler directive is being processed, the proper action is taken.

Any errors that are detected by the Assembler are displayed before the actual line containing the error is printed. If no source listing is being produced, error messages are still displayed to indicate that the assembly process did not proceed normally.

1.6 DISTRIBUTION DISK

The disk contains:

```
!asm11, !asm09, !asm05
Docs/
        !AsRef , !MCXref(Impression files), ReadMe, mxcmanual
Examples/
        hc11/
                l, m, x, p, s19, as11/
                        clkdrivr, mcx, system, vectors, forth11
        as05/
                l, m, x, p, s19, as05/
                        divide
```

CHAPTER 2

CODING ASSEMBLY LANGUAGE PROGRAMS

2.1 INTRODUCTION

Programs written in assembly language consist of a sequence of source statements. Each source statement consists of a sequence of ASCII characters ending with a carriage return. Appendix A contains a list of the supported character set.

2.2 SOURCE STATEMENT FORMAT

Each source statement may include up to four fields: a label (or "*" for a comment line), an operation (instruction mnemonic or assembler directive), an operand, and a comment.

2.2.1 Label Field

The label field occurs as the first field of a source statement. The label field can take one of the following forms:

1. An asterisk (*) as the first character in the label field indicates that the rest of the source statement is a comment. Comments are ignored by the Assembler, and are printed on the source listing only for the programmer's information.

2. A white space character (blank or tab) as the first character indicates that the label field is empty. The line has no label and is not a comment.

3. A symbol character as the first character indicates that the line has a label. Symbol characters are the upper or lower case letters a- z, digits 0-9, and the special characters, period (.), dollar sign ($), and underscore (_). Symbols consist of one to 15 characters, the first of which must be alphabetic or the special characters period (.) or underscore (_). All characters are significant and upper and lower case letters are distinct.

A symbol may occur only once in the label field. If a symbol does occur more than once in a label field, then each reference to that symbol will be flagged with an error.

With the exception of some directives, a label is assigned the value of the program counter of the first byte of the instruction or data being assembled. The value assigned to the label is absolute. Labels may optionally be ended with a colon (:). If the colon is used it is not part of the label but merely acts to set the label off from the rest of the source line. Thus the following code fragments are equivalent:

```
here:       deca
            bne     here
here        deca
            bne     here
```

A label may appear on a line by itself. The assembler interprets this as set the value of the label equal to the current value of the program counter.

The symbol table has room for at least 2000 symbols of length 8 characters or less. Additional characters up to 15 are permissible at the expense of decreasing the maximum number of symbols possible in the table.

2.2.2 Operation Field

The operation field occurs after the label field, and must be preceded by at least one white space character. The operation field must contain a legal opcode mnemonic or an assembler directive. Upper case characters in this field are converted to lower case before being checked as a legal mnemonic. Thus 'nop', 'NOP', and 'NoP' are recognized as the same mnemonic. Entries in the operation field may be one of two types:

Opcode. These correspond directly to the machine instructions. The operation code includes any register name associated with the instruction. These register names must not be separated from the opcode with any white space characters. Thus 'clra' means clear accumulator A, but 'clr a' means clear memory location identified by the label 'a'.

Directive. These are special operation codes known to the Assembler which control the assembly process rather than being translated into machine instructions.

2.2.3 Operand Field

The operand field's interpretation is dependent on the contents of the operation field. The operand field, if required, must follow the operation field, and must be preceded by at least one white space character. The operand field may contain a symbol, an expression, or a combination of symbols and expressions separated by commas.

The operand field of machine instructions is used to specify the addressing mode of the instruction, as well as the operand of the instruction. The following tables summarize the operand field formats for the various processor families. (NOTE: in these tables parenthesis "()" signify optional elements and angle brackets "<>" denote an expression is inserted. These syntax elements are present only for clarification of the format and are not inserted as part of the actual source program. All other characters are significant and must be used when required.)

2.2.3.1 M6805/M68HC05 Operand Syntax

For the M6805/68HC05, the operand formats are

Operand Format	*M6805/68HC05 Addressing Mode*
no operand	accumulator and inherent
<expression>	direct, extended, or relative
#<expression>	immediate
<expression>,X	indexed
<expression>,<expression>	bit set or clear
<expression>,<expression>,<expression>	bit test and branch

Details of the M6805/68HC05 addressing modes may be found in Appendix B.

2.2.3.2 M68HC11 Operand Syntax

For the M68HC11, the following operand formats exist:

Operand Format	*68HC11 Addressing Mode*
no operand	accumulator and inherent
<expression>	direct, extended, or relative
#<expression>	immediate
<expression>,X	indexed, X relative

<expression>,Y	indexed, Y relative
<expression>,<expression>	bit set or clear
<expression>,<expression>,<expression>	bit test and branch

Details of the 68HC11 addressing modes may be found in Appendix B.

The bit manipulation instruction operands are separated by spaces in this case since the HC11 allows bit manipulation instructions on indexed addresses. Thus a ',X' or ',Y' may be added to the final two formats above to form the indexed effective address calculation.

Details of the M68HC11 addressing modes may be found in Appendix B. The operand fields of assembler directives are described in Chapter 4.

2.2.3.3 Expressions.

An expression is a combination of symbols, constants, algebraic operators, and parentheses. The expression is used to specify a value which is to be used as an operand.

Expressions may consist of symbols, constants, or the character '*' (denoting the current value of the program counter) joined together by one of the operators: + - * / % & | ^ .

2.2.3.4 Operators.

The operators are the same as in c:

+	add
-	subtract
*	multiply
/	divide
%	remainder after division
&	bitwise and
|	bitwise or
^	bitwise exclusive or

Expressions are evaluated left to right and there is no provision for parenthesized expressions. Arithmetic is carried out in signed two's complement integer precision (that's 16 bits on the IBM PC).

2.2.3.5 Symbols.

Each symbol is associated with a 16-bit integer value which is used in place of the symbol during the expression evaluation. The asterisk (*) used in an expression as a symbol represents the current value of the location counter (the first byte of a multi-byte instruction).

2.2.3.6 Constants.

Constants represent quantities of data that do not vary in value during the execution of a program. Constants may be presented to the assembler in one of five formats: decimal, hexadecimal, binary, or octal, or ASCII. The programmer indicates the number format to the assembler with the following prefixes:

$	HEX
%	BINARY
@	OCTAL
'	ASCII

Unprefixed constants are interpreted as decimal. The assembler converts all constants to binary machine code and are displayed in the assembly listing as hex.

A decimal constant consists of a string of numeric digits. The value of a decimal constant must fall in the range 0-65535, inclusive. The following example shows both valid and invalid decimal constants:

VALID	INVALID	REASON INVALID
12	123456	more than 5 digits
12345	12.3	invalid character

A hexadecimal constant consists of a maximum of four characters from the set of digits (0-9) and the upper case alphabetic letters (A-F), and is preceded by a dollar sign ($). Hexadecimal constants must be in the range $0000 to $FFFF. The following example shows both valid and invalid hexadecimal constants:

VALID	INVALID	REASON INVALID
$12	ABCD	no preceding "$"
$ABCD	$G2A	invalid character
$001F	$2F018	too many digits

A binary constant consists of a maximum of 16 ones or zeros preceded by a percent sign (%). The following example shows both valid and invalid binary constants:

VALID	INVALID	REASON INVALID
%00101	1010101	missing percent
%1	%10011000101010111	too many digits
%10100	%210101	invalid digit

An octal constant consists of a maximum of six numeric digits, excluding the digits 8 and 9, preceded by a commercial at-sign (@). Octal constants must be in the ranges @0 to @177777. The following example shows both valid and invalid octal constants:

VALID	INVALID	REASON INVALID
@17634	@2317234	too many digits
@377	@277272	out of range
@177600	@23914	invalid character

A single ASCII character can be used as a constant in expressions. ASCII constants are preceded by a single quote ('). Any character, including the single quote, can be used as a character constant. The following example shows both valid and invalid character constants:

VALID	INVALID	REASON INVALID
'*	'VALID	too long

For the invalid case above the assembler will not indicate an error. Rather it will assemble the first character and ignore the remainder.

2.2.4 Comment Field

The last field of an Assembler source statement is the comment field. This field is optional and is only printed on the source listing for documentation purposes. The comment field is separated from the operand field (or from the operation field if no operand is required) by at least one white space character. The comment field can contain any printable ASCII characters.

2.3 ASSEMBLER OUTPUT

The Assembler output includes an optional listing of the source program and an object file which is in the Motorola S Record format. Details of the S Record format may be found in Appendix E.

The Assembler will normally suppress the printing of the source listing. This condition, as well as others, can be overridden via options supplied on the command line that invoked the Assembler.

Each line of the listing contains a reference line number, the address and bytes assembled, and the original source input line. If an input line causes more than 6 bytes to be output (e.g. a long FCC directive), additional bytes (up to 64) are listed on succeeding lines with no address preceding them.

The assembly listing may optionally contain a symbol table or a cross reference table of all symbols appearing in the program. These are always printed at the end of the assembly listing if either the symbol table or cross reference table options (Paragraph 4.8) are in effect. The symbol table contains the name of each symbol, along with its defined value. The cross reference table additionally contains the assembler-maintained source line number of every reference to every symbol. The format of the cross reference table is shown in Appendix D.

CHAPTER 3

RUNNING THE ASSEMBLERS

3.1 ASSEMBLER INSTALLING

!asm11/05 will require the following directories to be present:

root	$
level 1...n-1	...
level n	<project_name>
level n-1	<source>, <object>, l, m, x, p

It will write its temporary file to Wimp$ScrapDir and will complain if it's unset.

The accompanying example (on distribution disk) will clarify this subject.

3.2 ASSEMBLER INVOCATION

Double Click the !asmxx icon or drag it (RO3) to the icon bar. If you must compile several files at once, drag the first to the icon bar icon and the others to the pop-up window. The first file will give its name to the list, map and xref files, and to the proposed object, though this can be changed by dragging from its save window after successful compilation.

(If you really want to feel the thrill of command line, you can use the asmxx executables, depending on which processor family you wish to assemble code for. To run the assembler enter the following command line:

as* file1 (file2 . . .) (- option1 option2 . . .)

where file1, file2, etc are the names of the source files you wish to assemble. The source filenames may have extensions but the assembler does not check for any particular extensions.)

The options are one or more of the following (| means 'or, i.e. the two forms are equivalent):

-o <objname>	S19 file output name
-list|l	enables output listing (default: no listing)
-x|cre	enables the cross reference table generation
-sym|s	enables the symbol table generation
-cyc	enables cycle counting
-p <pages>	page every <page> lines
-crlf	add line feed to S19 file
-split	keep list, xref, sym files separated
-cpp	run C preprocessor
-noasm	do not execute the assembler if cpp is enabled (i.e. run cpp only)
-end	enables check for END directive in each file

The following cpp flags will be passed to the preprocessor:

-c, -e, -n, -x<parameter>, -d<parameter>, -i<parameter>, -s<parameter>, -u<parameter>

The object file created is written to disk when dragged from the save box, with the default name 'S19.FILENAME' where 'FILENAME' is the name of the first source file specified on the command line. Any errors are displayed on the screen and Throwback will point to the corresponding lines in the input files. The optional listing will be saved to a ''l.FILENAME' file for later examination. Same for Cross reference ''x.FILENAME' and symbol table ''m.FILENAME' . CPP output goes to the .p directory maintaining the input file name for every processed file.

Example: The command line

```
as11 system mcx clckdrivr vectors -list -x -sym -split -o test
```

runs the cpp on the source files system, mcx, clokdrivr, vectors. The preprocessed files will be written to the .p directory, where the as11reads them for compiling; object file is written to 'test'. A listing is created followed by a symbol table and cross reference which would all be written to the files l.system, m.system, x.system if the split option is specified, otherwise the list file will hold them all. Do not use the same name for input and output (-o) file from command line!

The same result would have more easily been obtained dragging 'system' to the !asm11 icon on the icon bar, dragging mcx, clokdrivr, vectors to the pop up menu, and clicking the appropriate icons. A file 's19.system' is automatically generated and suggested for saving at the end of the compilation.

3.3 ERROR MESSAGES

Error diagnostic messages are placed in the listing file just before the line containing the error. The format of the error line is:

```
Line_number:                Description of error
                            or
Line_number:                Warning ---- Description of error
```

Errors in pass one cause cancellation of pass two. Warning do not cause cancellation of pass two but are indications of a possible problem. Error messages are meant to be self-explanatory.

If more than one file is being assembled, the file name precedes the error:

```
File_name,Line_number:      Description of error
```

Some errors are classed as fatal and cause an immediate termination of the assembly. Generally this happens when a temporary file cannot be created or is lost during assembly.

Throwback will work according to its rules: remember to use a suitable editor! CPP does not support throwback (yet?).

CHAPTER 4

ASSEMBLER DIRECTIVES

4.1 INTRODUCTION

The Assembler directives are instructions to the Assembler, rather than instructions to be directly translated into object code. This chapter describes the directives that are recognized by the Freeware Assemblers. Detailed descriptions of each directive are arranged alphabetically. The notations used in this chapter are:

( ) Parentheses denote an optional element.

XYZ The names of the directives are printed in capital letters.

< > The element names are printed in lower case and contained in angle brackets.

All elements outside of the angle brackets '<>' must be specified as-is. For example, the syntactical element (<number>,) requires the comma to be specified if the optional element <number> is selected. The following elements are used in the subsequent descriptions:

<comment>	A statement's comment field
<label>	A statement label
<expression>	An Assembler expression
<expr>	An Assembler expression
<number>	A numeric constant
<string>	A string of ASCII characters
<delimiter>	A string delimiter
<option>	An Assembler option
<symbol>	An Assembler symbol
<sym>	An Assembler symbol
<sect>	A relocatable program section

In the following descriptions of the various directives, the syntax, or format, of the directive is given first. This will be followed with the directive's description.

4.2 BSZ - BLOCK STORAGE OF ZEROS

(<label>) BSZ <expression> (<comment>)

The BSZ directive causes the Assembler to allocate a block of bytes. Each byte is assigned the initial value of zero. The number of bytes allocated is given by the expression in the operand field. If the expression contains symbols that are either undefined or forward referenced (i.e. the definition occurs later on in the file), or if the expression has a value of zero, an error will be generated.

4.3 EQU - EQUATE SYMBOL TO A VALUE

<label> EQU <expression> (<comment>)

The EQU directive assigns the value of the expression in the operand field to the label. The EQU directive assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program. The expression cannot contain any forward references or undefined symbols. Equates with forward references are flagged with Phasing Errors.

4.4 FCB - FORM CONSTANT BYTE

```
(<label>)        FCB        <expr>(,<expr>,...,<expr>) (<comment>)
```

The FCB directive may have one or more operands separated by commas. The value of each operand is truncated to eight bits, and is stored in a single byte of the object program. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one or more of them can be null (two adjacent commas), in which case a single byte of zero will be assigned for that operand. An error will occur if the upper eight bits of the evaluated operands' values are not all ones or all zeros.

4.5 FCC - FORM CONSTANT CHARACTER STRING

```
(<label>)        FCC        <delimiter><string><delimiter> (<comment>)
```

The FCC directive has been slightly modified in this Arc porting. Originally, *any* ASCII printable characters could be used as delimiter: the assembler would pick the first non-blank after the FCC directive and expect the same character at the end of the string. As this could generate subtle bugs (suppose you wanted to write 'donald' and forgot the ('): the old assembler would assume 'onal' without complaining), I have changed it to be either ' or ". the symbol must be the same at the beginning and at the end of the string, i.e. you can not write 'qwerty" or "qwerty'. A bug that prevented the assembler from correctly recognizing the ';' char when enclosed in a FCC string has been fixed. The byte storage begins at the current program counter. The label is assigned to the first byte in the string. Any of the printable ASCII characters can be contained in the string. The string is specified between two identical delimiters which can be either ' or " characters.

Example:

```
LABEL1    FCC        'ABC'
```

assembles ASCII ABC at location LABEL1

4.6 FDB - FORM DOUBLE BYTE CONSTANT

```
(<label>)        FDB        <expr>(,<expr>,...,<expr>) (<comment>)
```

The FDB directive may have one or more operands separated by commas. The 16-bit value corresponding to each operand is stored into two consecutive bytes of the object program. The storage begins at the current program counter. The label is assigned to the first 16-bit value. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one or more of them can be null (two adjacent commas), in which case two bytes of zeros will be assigned for that operand.

4.7 FILL - FILL MEMORY

```
(<label>)        FILL       <expression>,<expression>
```

The FILL directive causes the assembler to initialize an area of memory with a constant value. The first expression signifies the one byte value to be placed in the memory and the second expression indicates the total number of successive bytes to be initialized. The first expression must evaluate to the range 0-255. Expressions cannot contain forward references or undefined symbols.

4.8 OPT - ASSEMBLER OUTPUT OPTIONS

```
 OPT              <option>(,<option>,...,<option>)    (<comment>)
```

The OPT directive is used to control the format of the Assembler output. The options are specified in the operand field, separated by commas. All options have a default condition. Some options can be initialized from the command line that invoked the Assembler, however the options contained in the source file take precedence over any entered on the command line. In the following descriptions, the parenthetical inserts specify "DEFAULT", if the option is the default condition. All options must be entered in lower case. The syntax of these options is not quite the same as that entered from command line: as OPT is rarely used (I guess), and I don't have much time, I decided to leave it as it was. See Appendix C

clcyc - Enable cycle counting in the listing. The total cycle count for that instruction will appear in the listing after the assembled bytes and before the source code

contclcontcyc - Restart cycle counting in the listing.

xlcre - Print a cross reference table at the end of the source listing. This option, if used, must be specified before the first symbol in the source program is encountered. The cross reference listing format may be found in Appendix D.

listll - Print the listing from this point on. A description of the listing format can be found in Appendix D.

noclnocyc - (DEFAULT) Disable cycle counting in the listing. If the "c" option was used previously in the program, this option will cause cycle counting to cease until the next "OPT c" statement.

nol - (DEFAULT) Do not print the listing from this point on. An "OPT l" can re-enable listing at a later point in the program.

symls - Print symbol table at end of source listing. The symbol table format can be found in Appendix D.

crlf -Add <CR> <LF> to S record

p50 - This may produce strange behaviour if used other than from command line, where it has a different syntax too. Avoid it.

4.9 ORG - SET PROGRAM COUNTER TO ORIGIN

```
        ORG             <expression>            (<comment>)
```

The ORG directive changes the program counter to the value specified by the expression in the operand field. Subsequent statements are assembled into memory locations starting with the new program counter value. If no ORG directive is encountered in a source program, the program counter is initialized to zero. Expressions cannot contain forward references or undefined symbols.

4.10 PAGE - TOP OF PAGE

```
        PAGE
```

The PAGE directive causes the Assembler to advance the paper to the top of the next page. If no source listing is being produced, the PAGE directive will have no effect. The directive is not printed on the source listing.

4.11 RMB - RESERVE MEMORY BYTES

```
        (<label>)       RMB         <expression>            (<comment>)
```

The RMB directive causes the location counter to be advanced by the value of the expression in the operand field. This directive reserves a block of memory the length of which in bytes is equal to the value of the expression. The block of memory reserved is not initialized to any given value. The expression cannot contain any forward references or undefined symbols. This directive is commonly used to reserve a scratchpad or table area for later use.

4.12 ZMB - ZERO MEMORY BYTES (same as BSZ)

```
        (<label>)        ZMB       <expression>        (<comment>)
```

The ZMB directive causes the Assembler to allocate a block of bytes. Each byte is assigned the initial value of zero. The number of bytes allocated is given by the expression in the operand field. If the expression contains symbols that are either undefined or forward references, or if the expression has a value of zero, an error will be generated.

4.13 BSS, CODE, DATA, AUTO

Block Storage Segment (RAM), Code Segment, Data Segment, Data Segment: unchecked!

4.14 END

The END directive, if enabled with the -end command line parameter, will be searched for in the file(s) (1 each file). If no END is encountered, a warning message is raised. Currently, the END directive can be encountered at any line of the file, i.e. it will not prevent the following lines from being compiled. It basically only suppresses the compiler warning, when enabled by -end, and the compiler will not complain if more than 1 END is encountered. Since this is not very useful, the -end option is off by default, i.e. no end checking.

CHAPTER 5

CPP — C Pre-Processor

5.1 INTRODUCTION

This is the description of the DECUS cpp as it comes with the distribution files.

CPP is originally written by Martin Minnow.

Please note that I ported it essentially to have #include and #define available under the Motoasm programs. All others features not checked.

5.2 SYNOPSIS

cpp [-options] [infile [outfile]]

5.3 DESCRIPTION

CPP reads a C source file, expands macros and include files, and writes an input file for the C compiler. If no file arguments are given, CPP reads from stdin and writes to stdout. If one file argument is given, it will define the input file, while two file arguments define both input and output files. The file name “-” is a synonym for stdin or stdout as appropriate.

5.4 OPTIONS

The following options are supported. Options may be given in either case.

5.4.1 -C

If set, source-file comments are written to the output file. This allows the output of CPP to be used as the input to a program, such as lint, that expects commands embedded in specially-formatted comments.

5.4.2 -Dname=value

Define the name as if the programmer wrote #define name value at the start of the first file. If“=value” is not given, a value of “1” will be used. On non-unix systems, all alphabetic text will not be forced to upper-case (this may be false under RISC OS i.e. RISC OS = unix).

5.4.3 -E

Always return “success” to the operating system, even if errors were detected. Note that some fatal errors, such as a missing #include file, will terminate CPP, returning “failure” even if the -E option is given.

5.4.4 -Idirectory

Add this directory to the list of directories searched for #include “...” and #include <...> commands. Note that there is no space between the “-I” and the directory string. More than one -I command is permitted. On non-Unix systems “directory” is forced to upper-case (this may be false under RISC OS i.e. RISC OS = unix).

5.4.5-N

CPP normally predefines some symbols defining the target computer and operating system. If -N is specified, no symbols will be predefined. If -N -N is specified, the “always present” symbols, __LINE__, __FILE__, and __DATE__ are not defined.

5.4.6 -Stext

CPP normally assumes that the size of the target computer’s basic variable types is the same as the size of these types of the host computer. (This can be overridden when CPP is compiled, however.) The -S option allows dynamic respecification of these values. “text” is a string of numbers, separated by commas, that specifies correct sizes.The sizes must be specified in the exact order:

char short int long float double

If you specify the option as “-S*text”, pointers to these types will be specified. -S* takes one additional argument for pointer to function (e.g. int (*)())

For example, to specify sizes appropriate for a PDP-11, you would write:

c s i l f d func

-S1,2,2,2,4,8,

-S*2,2,2,2,2,2,2

Note that all values must be specified.

5.4.7 -Uname

Undefine the name as if #undef name were given. On non-Unix systems, “name” will be forced to upper-case.

5.4.8 -Xnumber

Enable debugging code. If no value is given, a value of 1 will be used. (For maintenance of CPP only.)

5.5 PRE-DEFINED VARIABLES

When CPP begins processing, the following variables will have been defined (unless the -N option is specified):

Target computer (as appropriate):

pdp11, vax, M68000 m68000 m68k

Target operating system (as appropriate):

rsx, rt11, vms, unix

Target compiler (as appropriate):

decus, vax11c

The implementor may add definitions to this list. The default definitions match the definition of the host computer, operating system, and C compiler.

The following are always available unless undefined (or -N was specified twice):

__FILE__ The input (or #include) file being compiled (as a quoted string).

__LINE__ The line number being compiled.

__DATE__ The date and time of compilation as a Unix ctime quoted string (the trailing newline is removed). Thus,

printf("Bug at line %s,", __LINE__);

printf(" source file %s", __FILE__);

printf(" compiled on %s", __DATE__);

5.6 DRAFT PROPOSED ANSI STANDARD CONSIDERATIONS

The current version of the Draft Proposed Standard explicitly states that "readers are requested not to specify or claim conformance to this draft." Readers and users of Decus CPP should not assume that Decus CPP conforms to the standard, or that it will conform to the actual C Language Standard.

When CPP is itself compiled, many features of the Draft Proposed Standard that are incompatible with existing preprocessors may be disabled. See the comments in CPP's source for details.

The latest version of the Draft Proposed Standard (as reflected in Decus CPP) is dated November 12, 1984.

Comments are removed from the input text. The comment is replaced by a single space character. The -C option preserves comments, writing them to the output file.

The '$' character is considered to be a letter. This isa permitted extension.

The following new features of C are processed by CPP:

#elif expression (#else #if)

'\xNNN' (Hexadecimal constant)

'\a' (Ascii BELL)

'\v' (Ascii Vertical Tab)

#if defined NAME 1 if defined, 0 if not

#if defined (NAME) 1 if defined, 0 if not

#if sizeof (basic type)

unary +

123U, 123LU Unsigned ints and longs.

12.3L Long double numbers

token#token Token concatenation

#include token Expands to filename

The Draft Proposed Standard has extended C, adding a constant string concatenation operator, where

“foo” “bar”

is regarded as the single string “foobar”. (This does not affect CPP’s processing but does permit a limited form of macro argument substitution into strings as will be discussed.)

The Standard Committee plans to add token concatenation to #define command lines. One suggested implementation is as follows: the sequence “Token1#Token2” is treated as if the programmer wrote “Token1Token2”. This could be used as follows:

#line 123

#define ATLINE foo#__LINE__

ATLINE would be defined as foo123.

Note that “Token2” must either have the format of an identifier or be a string of digits. Thus, the string #define ATLINE foo#1x3

generates two tokens: “foo1” and “x3”.

If the tokens T1 and T2 are concatenated into T3, this implementation operates as follows:

1. Expand T1 if it is a macro.

2. Expand T2 if it is a macro.

3. Join the tokens, forming T3.

4. Expand T3 if it is a macro.

A macro formal parameter will be substituted into a string or character constant if it is the only component of that constant:

#define VECSIZE 123

#define vprint(name, size) \

printf(“name” “[” “size” “] = {n") ... vprint(vector, VECSIZE);

expands (effectively) to

vprint(“vector[123] = {n") ;

Note that this will be useful if your C compiler (here: assembler) supports the new string concatenation operation noted above. As implemented here, if you write

#define string(arg) “arg” ... string(“foo”) ...

This implementation generates “foo”, rather than the strictly correct “”foo“” (which will probably generate an error message). This is, strictly speaking, an error in CPP and may be removed from future releases.

5.7 ERROR MESSAGES

Many. CPP prints warning or error messages if you try to use multiple-byte character constants (non-transportable) if you #undef a symbol that was not defined, or if your program has potentially nested comments.

5.8 BUGS

The #if expression processor uses signed integers only.

I.e, #if 0xFFFFu < 0 may be TRUE.

As far as I know, the only incompatible aspect is that a ' or a " in a comment line, unless matched by a closing ' or ", will raise an "unmatched string" error. However, a correctly written asm program should not have unterminated strings after a FCC directive, so it's only a comment problem. Avoid 's and "s in comments or always use them in an even number each line. Should you need a single ' or " in a FCC directive, resort to use it ascii equivalent code.

As far as I know, the only incompatible aspect is that a ' or a " in a comment line, unless matched by a closing ' or ", will raise an "unmatched string" error. However, a correctly written asm program should not have unterminated strings after a FCC directive, so it's only a comment problem. Avoid 's and "s in comments or always use them in an even number each line. Should you need a single ' or " in a FCC directive, resort to use it ascii equivalent code.

APPENDIX A

CHARACTER SET

The character set recognized by the Freeware Assemblers is a subset of ASCII. The ASCII code is shown in the following figure. The following characters are recognized by the Assembler:

1. The upper case letters A through Z and lower case letters a through z.

2. The digits 0 through 9.

3. Five arithmetic operators: +, -, *, / and % (remainder after division).

4. Three logical operators: &, |, and ^.

5. The special symbol characters: underscore (_), period (.), and dollar sign ($). Only the underscore and period may be used as the first character of a symbol.

6. The characters used as prefixes for constants and addressing modes:

#	Immediate addressing
$	Hexadecimal constant
&	Decimal constant
@	Octal constant
%	Binary constant
'	ASCII character constant

7. The characters used as suffixes for constants and addressing modes:

,X	Indexed addressing
,Y	M68HC11 indexed addressing

8. Three separator characters: space, carriage return, and comma.

9. The character "*" to indicate comments. Comments may contain any printable characters from the ASCII set.

10. The special symbol backslash "\" to indicate line continuation. When the assembler encounters the line continuation character it fetches the next line and adds it to the end of the first line. This continues until a line is seen which doesn't end with a backslash or until the system maximum buffer size has been collected (typically greater or equal to 256).

ASCII CHARACTER CODES

BITS 4 to 6 --		0	1	2	3	4	5	6	7
	0	NUL	DLE	SP	0	@	P	`	p
B	1	SOH	DC1	:	1	A	Q	a	q
I	2	STX	DC2	!	2	B	R	b	r
T	3	ETX	DC3	#	3	C	S	c	s
S	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
0	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	′	7	G	W	g	w
T	8	BS	CAN	(	8	H	X	h	x
O	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
3	B	VT	ESC	+	;	K	[	k	{
	C	ff	FS	,	<	L	\	l	;
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	S1	US	/	?	O	_	o	DEL

APPENDIX B

ADDRESSING MODES

B.1 M6805/68HC05 ADDRESSING MODES.

INHERENT OR ACCUMULATOR ADDRESSING

The M6805 includes some instructions which require no operands. These instructions are self-contained, and employ the inherent addressing or the accumulator addressing mode.

IMMEDIATE ADDRESSING

Immediate addressing refers to the use of one byte of information that immediately follows the operation code in memory. Immediate addressing is indicated by preceding the operand field with the pound sign or number sign character (#). The expression following the # will be assigned one byte of storage.

RELATIVE ADDRESSING

Relative addressing is used by branch instructions. Branches can only be executed within the range -126 to +129 bytes relative to the first byte of the branch instruction. For this mode, the programmer specifies the branch address expression and places it in the operand field. The actual branch offset is calculated by the assembler and put into the second byte of the branch instruction. The offset is the two's complement of the difference between the location of the byte immediately following the branch instruction and the location of the destination of the branch. Branches out of bounds are flagged as errors by the assembler.

INDEXED ADDRESSING

Indexed addressing is relative to the index register. The address is calculated at the time of instruction execution by adding a one- or two-byte displacement to the current contents of the X register. The displacement immediately follows the operation code in memory. If the displacement is zero, no offset is added to the index register. In this case, only the operation code resides in memory. Since no sign extension is performed on a one-byte displacement, the offset cannot be negative. Indexed addressing is indicated by the characters ",X" following the expression in the operand field. The special case of ",X", without a preceding expression, is treated as "0,X". Some instructions do not allow a two-byte displacement.

DIRECT AND EXTENDED ADDRESSING

Direct and extended addressing utilize one (direct) or two (extended) bytes to contain the address of the operand. Direct addressing is limited to the first 256 bytes of memory. Direct and extended addressing are indicated by only having an expression in the operand field. Some instructions do not allow extended addressing. Direct addressing will be used by the Macro Assembler whenever possible.

BIT SET OR CLEAR

The addressing mode used for this type of instruction is direct, although the format of the operand field is different from the direct addressing mode described above. The operand takes the form <expression 1>, <expression 2>. <expression 1> indicates which bit is to be set or cleared. It must be an absolute expression in the range 0-7. It is used in generating the operation code. <expression 2> is handled as a direct address, as described above. Since the bit manipulation address is direct, only the first 256 locations may be operated on by bit manipulation operations.

BIT TEST AND BRANCH

This combines two addressing modes: direct and relative. The format of the operand is: <expression 1>, <expression 2>, <expression 3>. <expression 1> and <expression 2> are handled in the same manner as described above in "bit set or clear". <expression 3> is used to generate a relative address, as described above in "relative addressing".

B.2 M68HC11 ADDRESSING MODES.

PREBYTE

The number of combinations of instructions and addressing modes for the 68HC11 is larger than that possible to be encoded in an 8-bit word (256 combinations). To expand the opcode map, certain opcodes ($18, $1A, and $CD) cause the processor to fetch the next address to find the actual instruction. These opcodes are known as prebytes and are inserted automatically by the assembler for those instructions that require it.l In general the instructions contained in the alternate maps are those involving the Y register or addressing modes that involve the Y index register. Thus the programmer make the trade-off between the convenience of using the second index register and the additional time and code space used by the prebyte.

INHERENT OR ACCUMULATOR ADDRESSING

The M68HC11 includes some instructions which require no operands. These instructions are self-contained, and employ the inherent addressing or the accumulator addressing mode.

IMMEDIATE ADDRESSING

Immediate addressing refers to the use of one or more bytes of information that immediately follow the operation code in memory. Immediate addressing is indicated by preceding the operand field with the pound sign or number sign character (#). The expression following the # will be assigned one byte of storage.

RELATIVE ADDRESSING

Relative addressing is used by branch instructions. Branches can only be executed within the range -126 to +129 bytes relative to the first byte of the branch instruction. For this mode, the programmer specifies the branch address expression and places it in the operand field. The actual branch offset is calculated by the assembler and put into the second byte of the branch instruction. The offset is the two's complement of the difference between the location of the byte immediately following the branch instruction and the location of the destination of the branch. Branches out of bounds are flagged as errors by the assembler.

INDEXED ADDRESSING

Indexed addressing is relative one of the index registers X or Y. The address is calculated at the time of instruction execution by adding a one-byte displacement to the current contents of the X register. The displacement immediately follows the operation code in memory. If the displacement is zero, zero resides in the byte following the opcode. Since no sign extension is performed on a one-byte displacement, the offset cannot be negative. Indexed addressing is indicated by the characters ",X" following the expression in the operand field. The special case of ",X", without a preceding expression, is treated as "0,X".

DIRECT AND EXTENDED ADDRESSING

Direct and extended addressing utilize one (direct) or two (extended) bytes to contain the address of the operand. Direct addressing is limited to the first 256 bytes of memory. Direct and extended addressing are indicated by only having an expression in the operand field. Direct addressing will be used by the Assembler whenever possible.

BIT(S) SET OR CLEAR

The addressing mode used for this type of instruction is direct, although the format of the operand field is different from the direct addressing mode described above. The operand takes the form <expression 1> <expression 2> where the two expressions are separated by a blank. <expression 1> signifies the operand address and may be either a direct or an indexed address. When the address mode is indexed, <expression 1> is followed by ',R' where R is either X or Y. This allows bit manipulation instructions to operate across the complete 64K address map. <expression 2> is the mask byte. The bit(s) to be set or cleared are indicated by ones in the corresponding location(s) in the mask byte. The mask byte must be an expression in the range 0-255 and is encoded by the programmer.

BIT TEST AND BRANCH

This combines two addressing modes: direct or indexed and relative. The format of the operand is: <expression 1> <expression 2> <expression 3> where the expressions are separated by blanks. <expression 1> identifies the operand an may indicate either a direct or indexed address. Indexed addresses are signified with ',R' following the expression where R is either X or Y. <expression 2> is the mask byte. The bit(s) to be set or cleared are indicated by ones in the corresponding location(s) in the mask byte. The mask byte must be an expression in the range 0-255 and is encoded by the programmer. <expression 3> is used to generate a relative address, as described above in "relative addressing".

APPENDIX C

DIRECTIVE SUMMARY

A complete description of all directives appears in Chapter 4.

ASSEMBLY CONTROL

ORG Origin program counter

SYMBOL DEFINITION

EQU Assign permanent value

DATA DEFINITION/STORAGE ALLOCATION

BSZ Block storage of zero; single bytes

FCB Form constant byte

FCC Form constant character string

FDB Form constant double byte

FILL Initialize a block of memory to a constant

RMB Reserve memory; single bytes

ZMB Zero Memory Bytes; same and BSZ

LISTING CONTROL

OPT c	Enable cycle counting
OPT noc	Disable cycle counting
OPT contc	Re-enable cycle counting
OPT cre	Print cross reference table
OPT l	Print source listing from this point
OPT nol	Inhibit printing of source listing from this point
OPT s	Print symbol table
OPT crlf	Add <CR><LF> to S record
OPT p50	Turn on page flag
PAGE	Print subsequent statements on top of next page

APPENDIX D

ASSEMBLER LISTING FORMAT

The Assembler listing has the following format:

```
LINE#   ADDR  OBJECT CODE BYTES      [ # CYCLES]  SOURCE LINE
```

The LINE# is a 4 digit decimal number printed as a reference. This reference number is used in the cross reference. The ADDR is the hex value of the address for the first byte of the object code for this instruction. The OBJECT CODE BYTES are the assembled object code of the source line in hex. If an source line causes more than 6 bytes to be output (e.g. a long FCC directive), additional bytes (up to 64) are listed on succeeding lines with no address preceding them.

The # CYCLES will only appear in the listing if the "c" option is in effect. It is enclosed in brackets which helps distinguish it from the source listing. The SOURCE LINE is reprinted exactly from the source program, including labels.

The symbol table has the following format:

SYMBOL   ADDR

The symbol is taken directly from the label field in the source program. The ADDR is the hexadecimal address of the location referenced by the symbol.

The cross reference listing has the following format:

SYMBOL   ADDR   *LOC1 LOC2 LOC3 ...

The SYMBOL and ADDR are the same as above. The * indicates the start of the line reference numbers. The LOCs are the decimal line numbers of the assembler listing where the label occurs.

APPENDIX E

S-RECORD INFORMATION

E.1 INTRODUCTION

The S-record output format encodes program and data object modules into a printable (ASCII) format. This allows viewing of the object file with standard tools and allows display of the module while transferring from one computer to the next or during loads between a host and target. The S-record format also includes information for use in error checking to insure the integrity of data transfers.

E.2 S-RECORD CONTENT

S-Records are character strings made of several fields which identify the record type, record length, memory address, code/data, and checksum. Each byte of binary data is encoded as a 2-character hexadecimal number: the first character representing the high-order 4 bits, and the second the low-order 4 bits of the byte.

The 5 fields which comprise an S-record are:
TYPE RECORD LENGTH ADDRESS CODE/DATA CHECKSUM
The fields are defined as follows:

FIELD	CHARACTERS	CONTENTS
Type	2	S-record type - S1, S9, etc.
Record length	2	The count of the character pairs in the record, excluding the type and record length.
Address	4, 6, 8	The 2-, 3-, or 4-byte address at which the data field is to be loaded into memory.
Code/data	0-2n	From 0 to n bytes of executable code, memory loadable data, or descriptive information.
Checksum	2	The least significant byte of the one's complement of the sum of the values represented by the pairs of characters making up the record length, address, and the code/data fields.

Each record may be terminated with a CR/LF/NULL.

E.3 S-RECORD TYPES

Eight types of s-records have been defined to accommodate various encoding, transportation, and decoding needs. The Freeware assemblers use only two types, the S1 and S9:

S1 A record containing code/data and the 2-byte address at which the code/data is to reside.

S9 A termination record for a block of S1 records. The address field may optionally contain the 2-byte address of the instruction to which control is to be passed. If not specified, the first entry point specification encountered in the object module input will be used. There is no code/data field.

E.4 S-RECORD EXAMPLE

The following is a typical S-record module:

```
S1130000285F245F2212226A000424290008237C2A
S11300100002000800082629001853812341001813
S113002041E900084E42234300182342000824A952
S107003000144ED492
S9030000FC
```

The module consists of four code/data records and an S9 termination record.

The first S1 code/data record is explained as follows:
S1 S-record type S1, indicating a code/data record to be loaded/verified at a 2-byte address.
13 Hex 13 (decimal 19), indicating 19 character pairs, representing 19 bytes of binary data, follow.
00 Four-character 2-byte address field: hex address 0000, indicates location where the following data is to be loaded.
The next 16 character pairs are the ASCII bytes of the actual program code/data
2A Checksum of the first S1 record.
The second and third S1 code/data records each also contain $13 character pairs and are ended with checksums. The fourth S1 code/data record contains 7 character pairs.
The S9 termination record is explained as follows:
S9 S-record type S9, indicating a termination record.
03 Hex 03, indicating three character pairs (3 bytes) to follow.
00 Four character 2-byte address field, zeros.
00
FC Checksum of S9 record.