MOTOROLA

FREEWARE

8-BIT CROSS ASSEMBLERS

USER'S MANUAL

EDITED BY

KEVIN ANDERSON

FIELD APPLICATIONS ENGINEER

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER 1
GENERAL INFORMATION

1.1 INTRODUCTION

This is the user's reference manual for the IBM-PC hosted Motorola Freeware 8 bit cross assemblers. It 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. Those experienced with Motorola assembler products may wish to examine the file ASEMBLER.DOC available with the cross assemblers, which briefly describes the differences between these assemblers and earlier, non-pc based versions.

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 formating. (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 an IBM-PC or compatible and the target system is based on a Motorola 8-bit microprocessor (6800, 6801, 6803, 6805, 68HC05, 6809, or 68HC11).

The assemblers are the executable programs AS*.EXE where * is any of 0, 1, 4, 5, 9, or 11 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.

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 IBM-PC, XT, AT, PS-2, or true compatible. The assemblers may be run off of a floppy disk drive or they may be copied onto a hard drive for execution. DOS 2.0 or later is required.

1.4 ASSEMBLER PROCESSING

The Macro 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.

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 mneumonic 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 whitespace 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 whitespace character. The operation field must contain a legal opcode mneumonic or an assembler directive. Upper case characters in this field are converted to lower case before being checked as a legal mneumonic. Thus 'nop', 'NOP', and 'NoP' are recognized as the same mneumonic. 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 whitespace 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 whitespace 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 M6800/6801 Operand Syntax

The format of the operand field for M6800/6801 instructions is:

Operand Format	M6800/M6801 Addressing Mode
no operand	accumulator and inherent
<expression>	direct, extended, or relative
#<expression>	immediate
<expression>,X	indexed

Details of the M6800/6801 addressing modes may be found in Appendix B.

2.2.3.2 M6804/68HC Operand Syntax

For the M6804/68HC04, the following operand formats exist:

Operand Format	M6804/68HC04 Addressing Mode
no operand	accumulator and inherent
<expression>	direct, extended, or relative
#<expression>	immediate
<expression>	bit set or clear
<expression>,<expression>	bit test and branch
[<x> or <y>]	register indirect
<expression>,#<expression>	move indirect

Details of the M6804/68HC04 addressing modes may be found in Appendix B.

2.2.3.3 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.

2.2.3.4 M6809 Operand Syntax

For the M6809, the following operand formats are used:

Operand Format	M6809 Addressing Mode
no operand	accumulator and inherent
<expression>	direct, extended, or relative
#<expression>	immediate
<expression>,X	indexed
<<expression>	forced direct
><expression>	forced extended
<expression>]	extended indirect
<expression>,R	indexed
<<expression>,R	forced 8-bit offset indexed
><expression>,R	forced 16-bit offset indexed
[<expression>,R]	indexed indirect
<[<expression>,R]	forced 8-bit offset indexed indirect
>[<expression>,R]	forced 16-bit offset indexed indirect
Q+	auto increment by 1
Q++	auto increment by 2
[Q++]	auto increment indirect
-Q	auto decrement by
--Q	auto decrement by 2
[--Q]	auto decrement indirect
W1,[W2,...,Wn]	immediate

where R is one of the registers PCR, S, U, X, or Y, and Q is one of the registers S, U, X, or Y. Wi (i=1 to n) is one of the symbols A, B, CC, D, DP, PC, S, U, X, or Y.

Details of the M6809 addressing modes may be found in Appendix B.

2.2.3.5 M68HC11 Operand Syntax

For the M68HC11, the following operand formats exist:

Operand Format	M68HC11 Addressing Mode
no operand	accumulator and inherent
<expression>	direct, extended, or relative
#<expression>	immediate
<expression>,X	indexed with X register
<expression>,Y	indexed with Y register
<expression> <expression>	bit set or clear
<expression> <expression> <expression>	bit test and branch

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.

2.2.3.6 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: + - * / % & | ^ .

2.2.3.7 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-complement integer precision (that's 16 bits on the IBM PC).

2.2.3.8 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).

2.2.3.9 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 constan
ts:

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 inval
id 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 whitespace 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 INVOCATION

The Motorola Freeware Assembly programs are named as*.exe where '*' is any of 0, 1, 4, 5, 9, or 11 depending on which processor family you wish to assemble code for. For example, to generate M6800 code run the as0.exe program. To generate M68HC05 code run the as5.exe program, and so forth. 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 extension (however, do not use the .S19 extension since that is the extension of the object file created by the assembler. Its creation would overwrite the source file when it is written to the disk).

The options are one or more of the following:

l	enables output listing
no	disables output listing (default).
cre	enables the cross reference table generation
s	enables the symbol table generation
c	enables cycle counting
noc	disables cycle counting

The minus sign preceding the option should be separated from the last file name by a space. These options may also be indicated to the assembler by the use of the OPT directive in the source file. The OPT directive is described in Paragraph 4.8.

The object file created is written to disk and given the name 'FILENAME.S19' where 'FILENAME' is the name of the first source file specified on the command line. Any errors and the optional listing (if specified) are displayed on the screen. The listing and/or error messages may be saved to a file for later examination or printing by append an i/o redirection command to the command line. On the PC i/o redirection is indicated with the greater-than ('>') symbol followed by any new or existing file name.

Command line examples:

The command line

```
as5 myfile
```

would run the M6805/68HC05 assembler on the source file 'myfile'. The object file would be written to 'myfile.s19' and any errors would appear on the screen.

The command line

```
        as9 test.asm nexttest.s -l
```

would run the M6809 assembler on the source files 'test.asm' and 'nexttest.s'. The object file would be written to 'test.s19' and any errors and the assembly listing would appear on the screen.

The command line

```
        as9 test.asm nexttest.s -l cre s >test.lst
```

would run the M6809 assembler on the source files 'test.asm' and 'nexttest.s'. The object file would be written to 'test.s19'. A listing would be created followed by a symbol table and cross reference which would all be written to the file test.lst
.

3.2 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.

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
<reg list>	M6809 register list
<reg exp>	M6809 register expression

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 is used to store ASCII strings into consecutive bytes of memory. 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 any printable ASCII character. The first non-blank character after the FCC directive is used as the delimiter.

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.

c - 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.

cre - 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.

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

noc - (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-enble listing at a later point in the program.

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

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.

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
,PCR	M6809 indexed addressing
,S	M6809 indexed addressing
,U	M6809 indexed addressing
,Y	M6809 and 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).

11. For the M6809 Assembler, the character "<" preceding an expression to indicate direct addressing mode or 8-bit offset in indexed mode, and the character ">" preceding an expression to indicate extended addressing mode or 16-bit offset in indexed mode.

12. For the M6809 Assembler, the characters used to indicate auto increment and auto decrement in the indexed mode: +, ++, -, --.

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 M6800/M6801 ADDRESSING MODES.

INHERENT OR ACCUMULATOR ADDRESSING

The M6800 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 two 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 or two bytes of storage, depending on the instruction.

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-byte displacement (in the second byte of the instruction) to the current contents of the X register. Since no sign extension is performed on this 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.

B.2 M6804/M68HC04 ADDRESSING MODES.

INHERENT OR ACCUMULATOR ADDRESSING
The M6800 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 -15 to +16 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.

DIRECT AND EXTENDED ADDRESSING
Direct and extended addressing utilize byte to contain the address of the operand. Direct addressing is limited to the first 256 bytes of memory. Extended addressing concatenates the four least-significant bits of the opcode with the byte following the opcode to form a 12-bit address. 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.

SHORT DIRECT
Some opcodes allow 4 memory locations in data space ram ($80, $81, $82, and $83 to be referenced as part of the opcode. The opcode determines the data space RAM location, and the instruction is only one byte. The X and Y registers are at locations $80 and $81, respectively. An expression used with short direct addressing must not be forward referenced (that is its definition must occur before, not after this point in the file) and must equate to the range $80-$83.

BIT SET AND CLEAR
In the bit set/clear addressing mode, the bit to be set or cleared is part of the opcode. The byte following the opcode specifies the direct address of the byte which will have the bit set or cleared. Any bit in the 256 byte data space memory that can be written (with

the exception of the data direction registers) can be set or cleared with these two byte instructions.

BIT TEST AND BRANCH

The bit test and branch addressing mode is a combination of the direct addressing and relative addressing. The bit to be tested, and it condition (set or clear), is included in the opcode. The data space address of the byte to be tested is in the single byte immediately following the opcode byte and follows direct addressing rules. The third byte is sign extended by the processor during execution to form the 12-bit relative address which is added to the program counter if the condition is true. This allows branches based on any readable bit in the data space. The branch span is -125 to +130 from the opcode address. The branch target address is used by the programmer to signify the relative offset -- the assembler calculates the offset value. Branches out of bounds are flagged as errors by the assembler.

REGISTER INDIRECT

In the register indirect mode, the operand is at the address in data space pointed to by the contents of one of the indirect registers, X or Y. The particular indirect register is encoded in bit 4 of the opcode by the assembler. The assembler operand syntax for register indirect is

```
                    [<X> or <Y>]
```

MOVE IMMEDIATE

The MVI (move immediate) instruction has its own format:

```
         mvi   <expression 1>,#<expression 2>
```

where <expression 1> is a direct address and <expression 2> is the data value to be written.

MISCELLANEOUS SYNTAX ISSUES

The registers in the 6804/HC6804 are memory locations and have addresses assigned to them. The assembler has predefined

```
           a = A = $FF
           b = B = $80
           c = C = $81
```

This also means that for the '04 assembler clr x is equivalent to clrx since x is both a register and a memory location.

The '04 series has separate program and data spaces. There is no program memory in the range $10-$7F. Bytes assembled into that range will go into the data space.

B.3 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.4 M6809 ADDRESSING MODES.

INHERENT OR ACCUMULATOR ADDRESSING

The M6809 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 two 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 (#) -- i.e., #<expression>. The expression following the # will be assigned one or two bytes of storage, depending on the instruction. All instructions referencing the accumulator "A" or "B", or the condition code register "CC", will generate a one-byte immediate value. Also, immediate addressing used with the PSHS, PULS, PSHU, and PULU instructions generates a one-byte immediate value. Immediate operands used in all other instructions generate a two-byte value.

The register list operand does not take the form #<expression> but still generates one byte of immediate data. The form of the operand is:

```
R1,R2,...,Rn
```

where Ri (i=1 to n) is one of the symbols A, B, CC, D, DP, PC, S, U, X or Y. The number and type of symbols vary, depending on the specific instruction.

For the instructions PSHS, PULS, PSHU, and PULU, any of the above register names may be included in the register list. The only restriction is that "U" cannot be specified with PSHU or PULU, and "S" cannot be specified with PSHS or PULS. The one-byte immediate value assigned to the operand is calculated by the assembler and is determined by the registers specified. Each register name causes the assembler to set a bit in the immediate byte as follows:

Register	Bit
PC	7
U,S	6
Y	5
X	4
DP	3
B,D	2
A,D	1
CC	0

For the instructions EXG and TFR, exactly two of the above register names must be included in the register list. The other restriction is the size of the registers specified. For the EXG instruction, the two registers must be the same size. For the TFR instruction, the two registers must be the same size, or the first can be a 16-bit register and the second an 8-bit register. In the case where the transfer is from a 16-bit register to an 8-bit register, the least significant 8 bits are transferred. The 8-bit registers are A, B, CC, and DP. The 16-bit registers are D, PC, S, U, X, and Y. The one-byte immediate value assigned to the operand by the assembler is determined by the register names. The most significant four bits of the immediate byte contain the value of the first register name; the least significant four bits contain the value of the second register, as shown by the following table:.

Register	Value (hex)
D	0
X	1
Y	2
U	3
S	4
PC	5
A	8
B	9
CC	A
DP	B

RELATIVE ADDRESSING

Relative addressing is used by branch instructions. There are two forms of the branch instruction. The short branch 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 long branch can execute in the full range of addressing from 0000-FFFF (hexadecimal) because a two-byte offset is calculated by the assembler and put into the operand field 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.

DIRECT AND EXTENDED ADDRESSING

Direct and extended addressing utilize one (direct) or two (extended) bytes to contain the address of the operand. Direct and extended addressing are indicated by having only an expression in the operand field (i.e., <expression>). Direct addressing will be used whenever possible.

Regardless of the criteria described above, it is possible to force the Assembler to use the direct addressing mode by preceding the operand with the "<" character. Similarly, extended addressing can be forced by preceding the operand with the ">" character. These two operand forms are: <<expression> and ><expression>.

INDEXED ADDRESSING

Indexed addressing is relative to one of the index registers. The general form is <expression>,R. The address is calculated at the time of instruction execution by adding the value of <expression> to the current contents of the index register. The other general form is [<expression>,R]. In this indirect form, the address is calculated at the time of instruction execution by first adding the value of <expression> to the current contents of the index register, and then retrieving the two bytes from the calculated address and address+1. This two-byte value is used as the effective address of the operand. The allowable forms of indexed addressing are described below. In the description below, R refers to one of the index registers S, U, X, or Y.

The accumulator offset mode allows one of the accumulators to be specified instead of an <expression>. Valid forms are:.

```
                  <acc>,R and [<acc>,R]
```

where <acc> is one of the accumulators A, B, or D. This form generates a one-byte operand (post-byte only). When accumulator A or B is specified, sign extension occurs prior to adding the value in the accumulator to the index register.

The valid forms for the automatic increment/decrement mode are shown below. For each row, the three entries shown are equivalent.

R+	,R+	0,R+
-R	,-R	0,-R
R++	,R++	0,R++
--R	,--R	0,--R
[R++]	,R++]	[0,R++]
[--R]	[,--R]	[0,--R]

In this form, the only valid expression is 0. Like the accumulator offset mode, this form generates a one-byte operand (post-byte only).

The valid forms for the expression offset mode are:

```
R       ,R     <expression>,R
[R]     [,R]   [<expression>,R]
<R      <,R    <<expression>,R
<[R]    <[,R]  <[<expression>,R]
>R      >,R    ><expression>,R
>[R]    >[,R]  >[<expression>,R]
```

The "<" and ">" characters force an 8- or 16-bit offset, respectively, and are described below. If no expression is specified, or if an expression with a value of zero is specified, only the postbyte of the operand is generated. If an expression with a value in the range -16 to +15 is specified without indirection, a one- byte operand is generated which contains the expression's value, as well as the index register indicator. At execution time, the expression's value is expanded to 16 bits with sign extension before being added to the index register.

All other forms will generate a post-byte, as well as either a one- or two-byte offset which contains the value of the expression. The size of the offset is determined by the type and size of the expression. Expressions with values in the range -128 to +127 generate an 8-bit offset.   All other cases will result in a 16-bit offset being generated. In the case where an 8-bit offset is generated, the value is expanded to 16 bits with sign extension at execution time.

Regardless of the criteria described above, it is possible to force the Assembler to generate an 8-bit offset by preceding the operand with the "<" character. Similarly, a 16-bit offset can be forced by preceding the operand with the ">" character.

If the relative address calculated is not in the range -128 to +127, or if the expression references a symbol that has not yet been defined, a two-byte offset is generated after the post-byte. A one-byte offset is generated if the relative address is in the range -128 to +127.

Like the expression offset mode, a one-byte offset can be forced by preceding the operand with a "<". A ">" forces a two-byte offset. A byte overflow error is generated if a one-byte offset is forced when the relative address is not in the range -12
8 to +127.

The extended indirect mode has the form:

```
[<expression>]
```

Although extended indirect is a logical extension of the extended addressing mode, this mode is implemented using an encoding of the postbyte under the indexed addressing mode. A post-byte and a two-byte offset which contains the value of the expression is generated.

B.5 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 tradeoff 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 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
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, or 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 specifica tion 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 00	Four character 2-byte address field, zeroes.
FC	Checksum of S9 record.