M68MASR(D2)

SEPTEMBER 1979

M6800
M6801
M6805
M6809
MACRO ASSEMBLERS REFERENCE MANUAL

Second Edition

First Edition December 1978

TABLE OF CONTENTS

Page

CHAPTER 1

GENERAL INFORMATION

1.1 INTRODUCTION

The M6800 Macro Assembler is a program that processes source program statements written in M6800 assembly language. The Assembler translates these source statements into object programs compatible with the M6800 Linking Loader or the EXbug loader, and produces a listing of the source program. The M6800 Macro Assembler has been designed to operate on Motorola's 6800 Development System. The MDOS and tape versions of the M6800 Macro Assembler also support the M6801 instruction set. In addition, this manual describes the M6805 Macro Assembler and the M6809 Macro Assembler. Unless explicitly stated otherwise, all information pertaining to the M6800 Macro Assembler also pertains to the M6805 and M6809 Macro Assemblers. Although the Linking Loader is referred to as the M6800 Linking Loader, it supports M6800/6801, M6805, and M6809 programs.

The versions of the Macro Assembler described in this manual are:

```
RASM 3.00    (M6800/M6801 MDOS version)
RASM 2.2     (M6800 EDOS version)
RASM 2.2     (M6800/M6801 tape version)
RASM05 2.00 and 3.00   (M6805 MDOS version)
RASM09 3.01            (M6809 MDOS version)
```

Earlier versions of these products may not support all of the features described in this manual.

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, directives to the assembler, or macro instructions), symbolic names, operators, and special symbols.

The assembly language provides mnemonic operation codes for all machine instructions in the M6800 instruction set. The M6800 and M6801 instructions are defined and explained in the M6800/6801 Programming Reference Manual. The M6805 instructions are defined and explained in the M6805 Programming Reference Manual. The M6809 instructions are defined and explained in the M6809 Programming Reference Manual. 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. The assembly language also enables the programmer to define and use macro instructions which are used to replace a single statement with a predefined sequence of statements found in the macro definition.

1.3 OPERATING ENVIRONMENT

The minimum hardware requirements for the Macro Assembler include:

> Motorola 6800 Development system with EXbug monitor
> System console (keyboard and printer/display)
> M6800 EDOS version -- EXORdisk I, 16K RAM
> M6800/M6801 MDOS version -- EXORdisk II, 24K RAM
> M6800/M6801 Tape version -- Console reader/punch, 16K RAM
> M6805 MDOS version -- EXORdisk II, 24K RAM
> M6809 MDOS version -- EXORdisk II, 32K RAM

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 and macro tables. During the second pass, the object file is created (assembled) with reference to the tables 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. If there is no match, the macro definition table is scanned.

During the processing of a standard operation code mnemonic, the standard machine code is inserted into the object file. If a macro is being processed, the definition is expanded one line at a time and processed as a normal assembly language statement as defined above. 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. Errors are accumulated, and a total number of errors is printed at the end of each source listing. 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 5 fields: a sequence number, a label (or "*" for a comment line), an operation, an operand, and a comment.

2.2.1 Sequence Number

The sequence number field is an optional field provided as a programming convenience. The sequence number field starts at the beginning of the source line, and consists of up to five decimal digits. The value of the number must be less than 65536. Sequence numbers must be followed by a space. In MDOS versions of the Macro Assembler, sequence numbers will be automatically printed on the source listing. EDOS and tape versions of the Assembler will only print the sequence numbers under control of the OPT directive.

Although sequence numbers are optional, they must be consistently used or not used for an entire program. If the first source statement has a sequence number, then every succeeding source statment must also have a sequence number. If the first source statement does not have a sequence number, then no other source statement may be numered.

2.2.2 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 space 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 case letters A-Z, digits 0-9, and the special characters, period (.), dollar sign ($), and underscore (_). Symbols consist of one to six characters, the first of which must be alphabetic or the special character, period (.). Certain special symbols are reserved by the Assembler, and will cause an error to be generated if they appear in a label field. These reserved symbols are: A, B, and X. For the M6809 Macro Assembler, the following are also reserved symbols: CC, D, DP, PC, PCR, S, U, and Y. For the M6805 Macro Assembler, only A and X are reserved.

A symbol may occur only once in the label field unless it is used with the SET directive. 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 may be either relocatable or absolute. Chapter 3 contains a complete description of relocation in the Macro Assembler. In case the value is relocatable, the label is assigned the appropriate relocation attribute as well. Relocatable labels will have absolute values assigned to them during the link/load process performed with the M6800 Linking Loader.

Each unique label, undefined symbol, and external reference symbol in a program is allocated a ten-byte block in the symbol table. In addition, a ten-byte block is allocated for every four references to a symbol, if the cross reference option (paragraph 4.20) is in effect.

2.2.3 Operation Field

The operation field occurs after the label field, and must be preceded by at least one space. The operation field must contain a symbol. Thus, the rules governing labels apply to the operation field as well. Entries in the operation field may be one of three types:

Opcode — These correspond directly to the machine instructions. The operation code includes the "A" or "B" character for the accumulator specification. For compatibility with other M6800 assemblers, a single space may separate the operation code from the accumulator designator. For example, "LDA A" is the same as "LDAA". Although the M6809 Macro Assembler recognizes the above instruction forms (Appendix B.6), the proper form for the M6809 instruction "load accumulator A" is "LDA". The M6805 Macro Assembler does not recognize the opcode format that contains a space. In addition, only accumulator "A" is recognized.

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

Macro call — These indicate the selection of a previously defined macro which is to be inserted in place of the macro call.

The Assembler first searches for operation codes in an internal table of machine operation codes and assembler directives. If no match is found, the macro definition table is searched. Therefore, macros should not be given the names of existing instruction mnemonics, root mnemonics (such as ADD, SUB, EOR, etc.), or directives. If neither of the tables holds the specified operation code, an error message is printed. If code is being generated, three bytes of zeros are generated for an invalid operation code.

2.2.4 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 space. 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 format of the operand field for M6800 instructions is summarized in the following table:

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

For the M6805, the following additional operand formats exist:

Operand Format	M6805 Addressing Mode
<expression>,<expression>	bit set or clear
<expression>,<expression>,<expression>	bit test and branch

For the M6809, the following additional operand formats exist:

Operand Format	M6809 Addressing Mode
<<expression>	direct
><expression>	extended
[<expression>]	extended indirect
<expression>,R	indexed
<<expression>,R	8-bit offset indexed
><expression>,R	16-bit offset indexed
[<expression>,R]	indexed indirect
<[<expression>,R]	8-bit offset indexed indirect
>[<expression>,R]	16-bit offset indexed indirect
Q+	auto increment by 1
Q++	auto increment by 2
[Q++]	auto increment indirect
-Q	auto decrement by 1
--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.

The operand fields of assembler directives are described in Chapter 4. The operand fields of macros (Chapter 5) depend on the definition of the macro.

2.2.4.1 M6800/M6801 Addressing Modes. 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. The actual branch offset is 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 to externally referenced symbols or to symbols residing outside of the current program section are invalid.

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. Special cases of ",X" or "X" alone, without a preceding expression, are treated as "Ø,X". Since the displacement is a one-byte quantity, external references and addresses in sections other than BSCT and possibly ASCT are not valid.

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 Macro Assembler whenever possible. References to BSCT symbols (including external references to BSCT symbols) or to ASCT symbols with a value less than 256 will automatically be assembled with the direct addressing mode. If a directly-addressable symbol is referenced before it has been defined as being in BSCT (or ASCT less than 256), the instruction will be assembled with the extended addressing mode in order to avoid phasing errors. All other cases will result in extended addressing mode being used.

2.2.4.2 M6805 Addressing Modes. 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. Since the expression is one byte, external references and addresses in sections other than BSCT and possibly ASCT are not valid.

RELATIVE ADDRESSING

This addressing mode is the same as described for the M6800.

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. Special cases of ",X" or "X" alone, without a preceding expression, are treated as "Ø,X". Some instructions do not allow a two-byte displacement. When this is the case, external references and addresses in sections other than BSCT and possibly ASCT are not valid.

DIRECT AND EXTENDED ADDRESSING

The addressing mode is the same as described for the M6800 with one addition. Some instructions do not allow extended addressing. When this is the case, external references and addresses in sections other than BSCT and possibly ASCT are not valid.

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.

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".

2.2.4.3 M6809 Addressing Modes. 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 determined by the registers specified. Each register name sets 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

(Paragraph 4.24 contains a detailed explanation of immediate expressions with the PSH/PUL instructions.)

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 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. The actual branch offset is 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 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. Branches to externally referenced symbols or to symbols residing outside of the current program section are only valid for long branches.

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 by the M6809 Macro Assembler whenever possible. References to ASCT expressions with values having the most significant byte of the expression the same as the current value of the direct page pseudo register (Paragraph 4.27) will automatically be assembled with the direct addressing mode. References to BSCT symbols (including external references to BSCT symbols) will use the direct addressing mode only if the value of the direct page pseudo register is zero. If a symbol that follows the above rules is referenced before it has been defined, the instruction will be assembled with the extended addressing mode in order to avoid phasing errors. All other cases will result in extended addressing mode being used.

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>. There is no restriction on the latter form. It will always generate extended addressing. If direct addressing is forced, the following checks are made:

1. If the expression contains an external reference to a section other than BSCT, a relocation error will be generated.

2. If the expression contains symbols in sections other than BSCT, the expression will not be relocated by the M6800 Linking Loader. A warning message is generated to indicate this condition. Thus, the user must ensure that the direct page register at execution time is set up properly to accommodate direct addressing for such expressions.

3. If no error or warning message is generated as a result of checks 1 and 2, the most significant byte of the expression is compared with the direct page pseudo register. If they are not the same, a warning message is generated. Again, the user must ensure that the direct page register is set up at execution time.

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. Appendix B.5 describes the format of the post-byte (i.e., the byte immediately following the opcode) for each of the indexed addressing modes. 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 a non-relocatable expression with a value of zero is specified, only the post-byte of the operand is generated. If a non-relocatable 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. ASCT expressions with values in the range -128 to +127 generate an 8-bit offset. If an ASCT expression contains a symbol that is referenced before it has been defined, the instruction will be assembled using a 16-bit offset in order to avoid phasing errors. 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. Because of sign extention, even BSCT expressions generate 16-bit offsets. This eliminates the possibility of generating incorrect code in the case where a BSCT expression has a value of $80 or greater after relocation by the Linking Loader.

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. There is no restriction on the ">" form. It always generates a post-byte followed by a 16-bit offset. If an 8-bit offset is forced, the following checks are made:

1. If a relocatable expression contains symbols in section other than BSCT, a relocation error is generated. The user must beware that because of sign extension on eight bit offsets, a BSCT expression with a value of $8Ø or greater after relocation will give incorrect results.

2. If the expression is absolute but has a value outside of the range -128 to +127, a byte overflow error is generated.

The valid forms for the program counter relative mode are exactly the same as the expression offset mode, with the exception that the index register specification must be "PCR". However, the manner in which the offset is generated by the Assembler differs. The Assembler generates a relative address which is then used as the 8- or 16-bit offset following the post-byte. The relative address is the two's complement of the difference between the location of the byte immediately following the indexed instruction and the value of the expression. If the expression contains any external references or symbols residing outside of the current program section, a 16-bit offset is generated.

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 -128 to +127. A relocation error is generated if a one-byte offset is forced with an external symbol or one that contains another section reference.

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 post-byte under the indexed addressing mode. A post-byte is generated, as well as a two-byte offset which contains the value of the expression.

2.2.4.4 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 follow the conventional rules of algebra.

Expressions may contain relocatable or externally defined symbols. However, the following rules must be followed in order for the expression to be valid.

1. Relocatable symbols or expressions cannot be multiplied, divided, or operated on with the special two-character operators.

2. A relocation count is maintained for each program section represented within an expression. Adding a relocatable symbol causes the relocation count to be incremented; subtracting a relocatable symbol decrements the relocation count. After an expression has been evaluated, the following criteria must be met:

   a. All section counts except for one must be zero.

   b. The exception section must have a count of either zero or one or minus one.

   c. When an expression is used in conjunction with the one-byte immediate addressing mode, the indexed addressing mode, or with the FCB directive, all section counts except the BSCT count must be zero.

3. One or more external reference symbols may be added or subtracted without regard to section.

Only the least significant byte of an externally referenced symbol will be operated on by the M6800 Linking Loader when such symbols are used in conjunction with the immediate addressing mode (one byte immediate operand) or the indexed addressing mode. In the immediate addressing mode, only one externally referenced symbol is allowed.

2.2.4.5 Operators. The precedence of the various operators is as follows. Parenthetical expressions are evaluated first, with the innermost parentheses being processed before the outer ones. Next, the multiplication (*), division (/), and all two-character operators have precedence. Of lowest precedence are the addition (+) and subtraction (-) operators. Unary minus can only occur at the beginning of an expression or immediately before a left parenthesis. Unary minus is equivalent in evaluation to putting a zero directly before the minus sign. For example, the following expressions are all equivalent:

```
 -TAG1*INDEX+3
0-TAG1*INDEX+3
 -(TAG1*INDEX)+3
```

Operators of the same precedence are evaluated from left to right. All intermediate results in the computation of an expression are truncated to a 16-bit integer value. The result of an expression is also a 16-bit integer. Operators can operate on numeric constants, single character ASCII literals, and symbols.

In addition to the normal operators for multiplication, division, addition, and subtraction, the Assembler recognizes certain two-character operators. These operators are infix operators and have the same precedence as multiplication or division. Each two-character operator begins with an exclamation point (!) and takes two operands. The following two-character operators are defined:

!^	- exponentiation	The left operand is raised to the power specified by the right operand. If the right operand is zero, the resulting value will be "1", regardless of the value of the left operand.
!.	- logical AND	Each bit in the left operand is logically "ANDed" with the corresponding bit in the right operand.
!+	- inclusive OR	Each bit in the left operand is inclusively "ORed" with the corresponding bit in the right operand.
!X	- exclusive OR	Each bit in the left operand is exclusively "ORed" with the corresponding bit in the right operand.
!<	- shift left	The left operand is shifted to the left by the number of bits specified by the right operand. The left operand is zero-filled from the right.
!>	- shift right	The left operand is shifted to the right by the number of bits specified by the right operand. The left operand is zero-filled from the left.
!L	- rotate left	The left operand is rotated left by the number of bits specified by the right operand. The most significant bit is rotated into the least significant bit position of the left operand.
!R	- rotate right	The left operand is rotated right by the number of bits specified by the right operand. The least significant bit is rotated into the most significant bit position of the left operand.

2.2.4.6 Symbols. Each symbol is associated with a 16-bit integer value which is used in place of the symbol during the expression evaluation. Each symbol also has associated with it one of the following attributes:

1. Absolute attribute
2. Relocatable attribute
3. External reference (defined in another program)
4. Named Common name (cannot be used in expressions)
5. Undefined
6. SET symbol

An absolute, relocatable, or undefined symbol may also be used as an external definition (to be referenced by another program).

Certain symbols are special to the Assembler. These special symbols can only be used in expressions, and include the following:

*	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).
NARG	This symbol is only valid within a macro expansion. It takes on the value of the number of arguments that has been passed to the current level of expansion.

2.2.4.7 Constants. Constants represent quantities of data that do not vary in value during the execution of a program. The numeric constants can be in one of four bases: decimal, hexadecimal, binary, or octal.

A decimal constant consists of a string of numeric digits. The value of a decimal constant must fall in the range 0-65535, inclusive. Optionally, decimal constants may be preceded by the ampersand character (&). 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
&65201	67800	out of range (> 65535)

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 can also be designated by being succeeded by the letter "H". In this case, the first digit of the hexadecimal constant must be a numeric so that the constant can be distinguished from a symbol name. Hexadecimal constants must be in the range $0000 to $FFFF. The following example shows both valid and invalid hexadecimal constants:

VALID	INVALID	REASON INVALID
$12	ABCD	no preceding "$"
0ABCDH	$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 (%). Binary constants can also be represented by a series of ones and zeros succeeded by the letter "B". The following example shows both valid and invalid binary constants:

VALID	INVALID	REASON INVALID
%00101	1010101	missing percent
%1	%10011000101010111	too many digits
10100B	%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 can also be designated by ending in the letter "O" or "Q". Octal constants must be in the ranges @Ø to @177777. The following example shows both valid and invalid octal constants:

VALID	INVALID	REASON INVALID
@17634	@2317234	too many digits
377Q	@277272	out of range
1776000	23914Q	invalid character

Character constants can be used in expressions if they are single characters. Character 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

2.2.5 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 space. 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 optional object file which is in one of the following two formats: EXORciser-loadable format or relocatable format. For the MDOS versions of the Macro Assemblers, a third object file format exists -- MDOS loadable memory image. Appendix E contains the description of the source listing formats.

The Assembler will normally suppress the printing of the source listing, and select the generation of an object output file. These conditions, as well as others, can be overridden via options supplied on the command line that invoked the Assembler.

The assembly source program listing contains the original source statements, formatted for easier reading, as well as additional information which is generated by the Assembler. Most lines in the listing correspond directly to a source statement. Lines which do not correspond directly to source statements include: page headings, error messages, expansions of macro calls, or such directives as FCB, FCC, and FDB.

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 after the END directive if either the symbol table or cross reference table options (Paragraph 4.20) 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 E.3.

CHAPTER 3

RELOCATION AND LINKING

3.1 INTRODUCTION

"Relocation" refers to the process of binding a program to a set of memory locations at a time other than during the assembly process. For example, if subroutine "ABC" is to be used by many different programs, it is desirable to allow the subroutine to reside in any area of memory. One way of repositioning the subroutine in memory is to change the "ORG" directive's operand field at the beginning of the subroutine, and then to re-assemble the routine. A disadvantage of this method is the expense of re-assembling ABC. An alternative to multiple assemblies is to assemble ABC once, producing an object module which contains enough information so that another program (the M6800 Linking Loader) can easily assign a new set of memory locations to the module. This scheme offers the advantages that re-assembly is not required, the object module is substantially smaller than the source program, relocation is faster than re-assembly, and relocation can be handled by the Linking Loader (rather than editing the source program and changing the ORG directive).

In addition to program relocation, the Linking Loader must also resolve inter-program references. For example, the other programs that are to use subroutine ABC must contain a jump-to-subroutine instruction to ABC. However, since ABC is not assembled at the same time as the calling program, the Assembler cannot put the address of the subroutine into the operand field of the subroutine call. The Linking Loader, however, will know where the calling program resides and, hence, can resolve the reference to the call to ABC. This process of resolving inter-program references is calling "linking".

The relocation and linking scheme was developed to provide the following capabilities:

1. Program relocation
2. Multiple program linking
3. Easy development of programs for RAM/ROM environment
4. Easy specification of any addressing mode
5. Specification of uninitialized, blank common
6. Specification of initialized, named common

Program sections provide the basis of the relocation and linking scheme. There are five different sections. They are described below.

ASCT, or absolute section, is a non-relocatable section. There may be a limited number of absolute sections in a user's program. These sections are used to allocate or initialize memory locations that are assigned by the programmer rather than by the M6800 Linking Loader. ASCT can be used to define the locations of PIA's or ACIA's, for example.

BSCT, or base section, is a relocatable section. There is only one base section. The M6800 Linking Loader assigns portions of the base section to each module that requires space in BSCT. The base section is generally used for variables that are to be accessed using the direct addressing mode. BSCT is restricted to memory locations 0-255, inclusive (decimal).

CSCT, or blank common, is a relocatable section. There is only one blank common section. CSCT is similar to blank common used in FORTRAN. The blank common section cannot be initialized.

DSCT, or data section, is a relocatable section. There is only one data section. The M6800 Linking Loader assigns portions of this section to each program that requires space in DSCT. DSCT is generally used to contain variables which are in RAM and are to be accessed using the extended addressing mode.

PSCT, or program section, is a relocatable section. There is only one program section. PSCT is similar to DSCT. However, it is generally used to contain program instructions. The use of DSCT and PSCT facilitates creation of programs that reside in ROM but access variables in RAM.

Uninitialized, blank common is placed into CSCT as described above. At times, however, it is convenient to have several common areas, each of which may be initialized. Therefore, the concept of named common was included in the M6800 relocation and linking scheme. Named common can be specified in either BSCT, DSCT, or PSCT. The size of the named common area that is allocated will be the largest of the named common sizes from the program modules that reference it. A named common block must reside wholly within a single section.

For a complete description of the M6800 Linking Loader, the M6800 Linking Loader Reference Manual should be consulted.

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 Macro Assembler. Detailed descriptions of each directive are arranged alphabetically. The notations used in this chapter are:

{ } Contains a list of elements, one of which must be selected. Each choice will be separated by a vertical bar. For example, {IFC¦IFNC} indicates that either IFC or IFNC must be selected.

[ ] Contains an optional element. If one of a series of elements may be selected, the available list of choices will be contained within the brackets. Each choice will be separated by a vertical bar. For example, [BSCT¦DSCT¦PSCT] indicates that either BSCT, DSCT, or PSCT may be selected.

XYZ The names of the directives are printed in capital letters. The required parts of directive operands will also be printed in capital letters. 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 element names are printed in lower case and contained in angle brackets. 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 ASCT - ABSOLUTE SECTION

ASCT [<comment>]

The ASCT directive causes the program counter to be restored to the address following the address of the last byte previously allocated to an absolute section (or to zero if ASCT is used for the first time). The program counter becomes absolute, and subsequent object code will not be relocated. The ASCT directive may only be used if a program is being assembled with the relocatable option (OPT REL).

4.3 BSCT - BASE SECTION

BSCT [<comment>]

The BSCT directive causes the program counter to be restored to the address following the address of the last byte previously allocated to the base section (or to zero if BSCT is used for the first time). The program counter becomes relocatable within the base section. All symbols that are defined in BSCT will be accessed using the direct addressing mode if the symbols are defined prior to being referenced. With the M6809 Macro Assembler, direct addressing in BSCT is only used if the direct page pseudo register is set to zero (Paragraph 4.27). BSCT cannot be larger than 256 (decimal) bytes. The BSCT directive may only be used if the program is being assembled with the relocatable option (OPT REL).

4.4 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 external references or forward references, or if the expression has a value of zero, an error will be generated.

4.5 COMM - NAMED COMMON SECTION

<label> COMM {BSCT ¦ DSCT ¦ PSCT} [<comment>]

The COMM directive causes the program counter to be restored to the address following the address of the last byte previously allocated to the named common section specified by the <label> field (or to zero if <label> is used for the first time). The program counter becomes relocatable, and subsequent object code will be relocated within the named common section. The COMM directive is one of the directives that assigns a value other than the program counter to the label.

Named common allows the definition of a group of symbols that are to occupy the same area of memory in each of several programs that are to reside in different areas of memory. Each symbol is defined as a relative offset from the beginning of the named common section. When the relocatable programs are link/loaded via the M6800 Linking Loader, each reference to a named common section is relocated by the starting address assigned to the section by the Linking Loader. The Linking Loader allocates enough memory to accommodate the largest named common section defined by any of the linked programs.

The COMM directive's <label> field becomes the name of the named common section. This symbol cannot be used in any subsequent Assembler expressions. The <label> can only appear with other COMM directives within the program. The operand of the COMM directive defines what addressing mode will be used to reference symbols that are defined in the named common section. Subsequent references to the named common section identified by <label> must have the same operand field.

The COMM directive may only be used if the program is being assembled with the relocatable option (OPT REL).

4.6 CSCT - BLANK COMMON SECTION

CSCT [<comment>]

The CSCT directive causes the program counter to be restored to the address following the address of the last byte previously allocated to the blank common section (or to zero if CSCT is being used for the first time). The program counter becomes relocatable, and subsequent memory bytes reserved will be relocated within the blank common section. No initialization (object code) of CSCT is allowed. Only the RMB directive can be used to allocate storage. All symbols defined with CSCT will be accessed with the extended addressing mode. With the M6809 Macro Assembler, direct addressing can be used to access symbols in CSCT if the operand field in which they are referenced is preceded with a "<" (Paragraph 2.2.4.3). The CSCT directive may only be used if the program is being assembled with the relocatable option (OPT REL).

4.7 DSCT - DATA SECTION

DSCT [<comment>]

The DSCT directive causes the program counter to be restored to the address following the address of the last byte previously allocated to the data section (or to zero if DSCT is being used for the first time). The program counter becomes relocatable, and subsequent object code will be relocated within the data section. All symbols defined within DSCT will be accessed with the extended addressing mode. With the M6809 Macro Assembler, direct addressing can be used to access symbols in DSCT if the operand field in which they are referenced is preceded with a "<" (Paragraph 2.2.4.3). The DSCT directive may only be used if the program is being assembled with the relocatable option (OPT REL).

4.8 END - END OF SOURCE PROGRAM

END [<expression> [<comment>]]

The END directive indicates that the logical end of the source program has been encountered. Any statements following the END directive are ignored. If the END directive is not encountered before the physical end of the source file is found, an error will be generated. However, this error is only a warning. The optional expression in the operand field can be used to specify the starting execution address of the program.

4.9 ENDC - END OF CONDITIONAL ASSEMBLY

ENDC [<comment>]

The ENDC directive is used to signify the end of the current level of conditional assembly (Paragraph 4.17). Conditional assembly directives can be nested to a depth of eight.

4.10 ENDM - END OF MACRO DEFINITION

ENDM [<comment>]

The ENDM directive is used in a macro definition (Paragraph 4.18). Its presence indicates the end of the macro definition.

4.11 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 is one of the directives that 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 external references, forward references, or undefined symbols. The expression may, however, be relocatable.

4.12 FAIL - PROGRAMMER GENERATED ERROR

FAIL [<string>]

The FAIL directive will cause an error message to be printed by the Assembler. The total error count will be incremented as with any other error. The FAIL directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the error has been printed. The <string> can be optionally specified to describe the nature of the generated error.

4.13 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. The expressions may be relocatable with respect to BSCT, or may contain BSCT external references. However, all other external references or relocatable symbol types are invalid.

4.14 FCC - FORM CONSTANT CHARACTER STRING

[<label>] FCC <number>,<string> [<comment>]

or

[<label>] FCC <delimiter><string><delimiter> [<comment>]

The FCC directive is used to store ASCII strings into consecutive bytes of memory. Any of the printable ASCII characters can be contained in the string. The FCC directive has two formats. The first format requires that <number> be a decimal constant in the range 1-255. The comma is required after the decimal constant. The <number> specifies the number of characters contained in <string>, which begins immediately after the comma. Should <number> be greater than the number of characters in the string (e.g., carriage return encountered in line before specified number of characters are found), then spaces will be inserted to fill the remainder of the string.

The second format of the FCC directive specifies the string between two identical delimiters. The delimiters can be any printable ASCII character. The first non-blank character after the FCC directive will be used as the delimiter. Thus, if the delimiter happens to be a decimal digit, the first character of the string cannot be a comma.

4.15 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. 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.

4.16 IDNT - RELOCATABLE IDENTIFICATION RECORD

IDNT <string>

The IDNT directive is used to create an identification record for the relocatable object module. The <string> can be any printable ASCII characters. The end of <string> is the terminating carriage return. This identification record can subsequently be displayed by the M6800 Linking Loader during the link/load process. The IDNT directive only has meaning when the program is being assembled with the relocatable option (OPT REL).

4.17 IFxx - CONDITIONAL ASSEMBLY DIRECTIVES

{IFC¦IFNC} <string 1>,<string 2>

or

{IFEQ¦IFGE¦IFGT¦IFLE¦IFLE¦IFNE} <expression> [<comment>]

The IFxx directives are used to conditionally assemble a section of a source program. The portion of the source program following the IFxx directive up to the next ENDC directive is conditionally assembled, depending on the result of the string comparisons (first form) or depending on the value of the expression in relation to the condition (the second form).

The IFC directive will cause the subsequent statements to be assembled if the two strings compare. The IFNC directive will cause the subsequent statements to be assembled if the two strings do not compare. In either case, if the condition is not met (comparison in the first case, and no comparison in the second case), the subsequent statements will be excluded from the assembly. The beginning of <string 1> is the first non-blank, non-comma character after the IFxx directive. The end of <string 1> is the last character before the first comma. The beginning of <string 2> is the first character after the first comma. The end of <string 2> is the last character before the end of the source line. Thus, if the first form of the IFxx directive is used, no comment can appear on the source statement. Both <string 1> and <string 2> can be null. <string 1> will be null if only a comma is specified after the IFxx directive. <string 2> will be null if only a carriage return is found after the comma.

If the second form of the IFxx directive is used, the subsequent statements will be assembled if the expression is:

IFEQ -- equal to zero
IFGE -- greater than or equal to zero
IFGT -- greater than zero
IFLE -- less than or equal to zero
IFLT -- less than zero
IFNE -- not equal to zero

If the condition is not met, the subsequent statements will be excluded from the assembly.

Conditional assembly directives can be nested to a depth of eight. Chapter 5 contains a complete description of the IFxx directives.

4.18 MACR - MACRO DEFINITION

<label> MACR [C] [<comment>]

The MACR directive is used to define a macro. All statements following the MACR directive up to the next ENDM directive become a part of the macro definition. The required label is the symbol by which the macro will subsequently be called. The MACR directive is one of the directives that assigns a value other than the program counter to the label. Macro names must not be names of existing instruction mnemonics, root mnemonics (e.g., SUB, EOR, ADD, etc.), or Assembler directives. The operand field may optionally contain the letter "C". If the C is present, then all comment lines (lines with an asterisk in column 1) will be retained in the macro definition. If the C is not specified, then all comment lines will be excluded from the definition. Since macro definitions are stored in memory, ommitting the C will reduce the memory requirements of the macro definition. Macro definitions may not be nested -- that is, another MACR directive cannot be encountered before the ENDM directive. Chapter 5 contains a complete description of macros.

4.19 NAM - ASSIGN PROGRAM NAME

NAM [<string> [<comment>]]

The NAM directive is generally used as the first statement of an assembly language program. Its use, however, is optional, and more than one NAM directive can be used in a program. The <string> specified will be printed on the heading line of each page of the source listing. It will be used as the name in the S0 record if an absolute EXORciser-loadable program is being created; or it will be the name of the relocatable program module (displayed by the M6800 Linking Loader) if relocation has been specified. The <string> consists of a maximum of six printable ASCII characters.

4.20 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 are not reset to their default conditions at the end of pass one. Some options are allowed to have the prefix "NO" attached to them, which then reverses their meaning. Depending on the version of the Macro Assembler, most options can be initialized from the command line that invoked the Assembler. In the following descriptions, the parenthetical inserts specify "DEFAULT", if the option is the default condition, and "RESET", if the option is reset to its default condition at the end of pass one. The text in the OPTION column of the following table indicates the minimum characters that are required to identify the option. Additional characters can be appended to the end of an option to make it more readable, depending on programmer preference. For example, CL can be CLIST, NOG can be NOGEN, L can be LIST, U can be UNASSEMBLE, etc.

OPTION	MEANING
ABS	Select absolute MDOS-loadable object output (non-relocatable). All relocatable directives are invalid if this option is specified. The "REL" and "LOAD" options are invalid if this option is used. This option is only supported on MDOS versions of the Macro Assemblers.
CL (DEFAULT, RESET)	Print the conditional assembly directives.
NOCL	Do not print the conditional assembly directives.
CMO	Only valid with M6805 Macro Assembler. Allow CMOS instructions STOP and WAIT.
NOCMO (DEFAULT, RESET)	Only valid with M6805 Macro Assembler. Do not allow CMOS instructions STOP and WAIT.
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.
G	Print the code generated for multiple operands of the FCB, FCC, and FDB directives.

NOG (DEFAULT, RESET) — Do not print the code generated for multiple operands of the FCB, FCC, and FDB directives.

L — Print the listing from this point on. The "L" option causes an internal list counter to be incremented. As long as the list counter is greater than zero, the subsequent source listing will be printed. If the source listing is not specified on the command line that invoked the Assembler, the L option has no effect when encountered within the source program.

NOL (DEFAULT, RESET) — Do not print the listing from this point on (including the OPT NOL directive). The "NOL" option causes an internal list counter to be decremented. As long as the list counter is less than or equal to zero, the subsequent source listing will not be printed. Thus, the NOL and L options can be nested. For example:

```
        OPT     NOL
MAC1    MACR
        OPT     NOL
        .
        .
        OPT     L
        ENDM
        OPT     L
```

The listing will be turned off with the first NOL option causing the macro definition to be omitted from the source listing. The last L option will cause the listing to be turned on again, resuming the printing of the source program. The NOL and L options within the body of the macro are used to suppress printing of the macro at expansion time, regardless of the state of the "MEX" option.

LLE=<number> — Change the number of characters to be printed per line to the decimal number specified. The default value is 72; the minimum value is 50; and the maximum value is 120.

LOAD (DEFAULT) — Select absolute EXORciser-loadable object output (non-relocatable). All relocatable directives are invalid if this option is specified. The "REL" and "ABS" options are invalid if this option is used.

M — Direct object output into memory. This option cannot be used in conjunction with the "REL" option. The Assembler will only allow memory to be used for the object output that is beyond the end of the available contiguous memory. If an error occurs while placing object code into memory (non-existent memory or Assembler memory), an error message will be displayed and the "M" option will be disabled. This option is not to be confused with "M" command line option of the MDOS version of the Macro Assembler (see Appendix G.1).

MC (DEFAULT, RESET)	Print macro calls
NOMC	Do not print macro calls.
MD (DEFAULT, RESET)	Print macro definitions.
NOMD	Do not print macro definitions.
MEX	Print macro expansions
NOMEX (DEFAULT, RESET)	Do not print macro expansions
O (DEFAULT)	Create output module. Since this option is normally selected, it need not be specified. It instructs the Assembler to create an object module (either in memory or in a file). This option can only be used once within a source program.
NOO	Do not create object output module. This option is used to suppress creation of an output module. This option will suppress the creation of the object module even if the creation of one was specified on the command line that invoked the Assembler.
P=<number>	Change the number of source statements printed per page to the decimal number specified. The default value is 58; the minimum value is 10; and the maximum value is 255.
NOP	Suppress paging; ignore PAGE directives and do not print headings or page numbers.
REL	Select relocatable object output. This option indicates that the assembly is to be done in the relocatable mode. Any object code produced will be in the relocatable record format. All relocatable directives are valid if this option is specified. The "REL" option should be placed before any statement in the source file (other than NAM directive or comment lines). The REL option is invalid if used with the LOAD, ABS, or M options.
S	Print symbol table at end of source listing. This option has no effect if the "CRE" option is used.
SE	Print the user-supplied sequence numbers in the right margin of the source listing. This option is ignored in the MDOS version of the Macro Assembler which automatically prints the user-supplied sequence numbers. Only the EDOS and tape versions of the Macro Assembler respond to this option.
U	Print the unassembled lines skipped due to failure to satisfy the condition of a conditional assembly directive.
NOU (DEFAULT, RESET)	Do not print the lines excluded from the assembly due to a conditional assembly directive.

W (DEFAULT, RESET) — Only valid with M6809 Macro Assembler. Print warning messages.

NOW — Only valid with M6809 Macro Assembler. Do not print warning messages.

ZØ1 — Only valid with MDOS and tape versions of the M6800 Macro Assembler. Allow M6801 instruction mnemonics to be assembled. This option permits the Assembler to recognize valid M6801 instruction menmonics (Appendix B.2). If "Z01" is specified, the M6800 mnemonics will still be recognized and assembled properly. In addition, the object code for any M6801 instructions will also be generated correctly. This option can be used more than once in a program.

NOZØ1 (DEFAULT, RESET) — Only valid with MDOS and tape versions of the M6800 Macro Assembler. Disallow M6801 instruction mnemonics. If this option is used in conjunction with the ZØ1 option, all subsequent M6801 instructions (until another ZØ1 option) will cause errors to be generated.

4.21 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. If the program is being assembled with the relocatable option (OPT REL), the default program counter value is zero and in PSCT. Expressions in the operand field can be relocatable. If they are, they may change the program counter section, as well as the program counter's value. Expressions cannot contain external references, forward references, or undefined symbols.

4.22 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.

4.23 PSCT - PROGRAM SECTION

PSCT [<comment>]

The PSCT directive causes the program counter to be restored to the address following the address of the last byte previously allocated to the program section (or to zero if PSCT is used for the first time). The program counter becomes relocatable, and subsequent object code will be relocated within the program section. All symbols defined within PSCT will be accessed with the extended addressing mode. Direct addressing of PSCT symbols is not possible, except with the M6809 Macro Assembler where direct addressing can be used to access symbols in PSCT if the operand field in which they are referenced is preceded with a "<" (Paragraph 2.2.4.3). The PSCT directive may only be used if the program is being assembled with the relocatable option (OPT REL).

4.24 REG - DEFINE REGISTER LIST

<label> REG <reg list> [<comment>]

The REG directive is only supported by the M6809 Macro Assembler. It assigns a value associated with a register list to the label. The REG directive is one of the directives that assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program. <reg list> must be of the form:

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. An error message is generated if both U and S are specified. A warning occurs if the same register is specified more than once. Register D is the same as registers A and B.

Although <label> may be used in any expression, its value is only meaningful when used with the instructions PSHU, PULU, PSHS, and PULS. The operand for these instructions can take one of two forms:

{PSHU¦PULU¦PSHS¦PULS} <reg list>

or

{PSHU¦PULU¦PSHS¦PULS} #<reg exp>

<reg list> is in the same format as defined above. An error message is generated if the register list contains a "U", and the instruction is PSHU or PULU. Similarly, an error occurs if the register list contains an "S", and the instruction is PSHS or PULS. <reg exp> is of the form:

<sym 1>[!+<sym 2>!+...!+<sym n>]

where <sym i> (i=1 to n) must be defined by the REG directive. An error occurs if a PSHU/PULU instruction is followed by a <reg exp> that contains a symbol previously defined with the REG directive that contained a U in the register list. A similar check is made for PSHS/PULS and S.

<u>Valid Examples</u>

```
ALLREG REG   A,B,CC,DP,X,Y,U,PC
REGXY  REG   X,Y
REGAB  REG   A,B
       PSHS  #ALLREG
       PSHU  #REGXY!+REGAB
```

<u>Invalid Examples</u>

```
REGUS  REG   U,S          can't specify both U and S
REGU   REG   U
       PSHU  #REGU        can't push U reg. onto U reg.
REGLST REG   A,B,D        duplicate reg. name warning
       PSHS  #REGU!+REGU  duplicate reg. name warning
```

4.25 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 external references, forward references, or undefined symbols. The value of the expression cannot be relocatable. The RMB directive is the only storage allocation operation that is allowed in the blank common section, CSCT.

4.26 SET - SET SYMBOL TO A VALUE

<label> SET <expression> [<comment>]

The SET directive assigns the value of the expression in the operand field to the label. The SET directive functions like the EQU directive. However, labels defined via the SET directive can have their values redefined in another part of the program (but only through the use of another SET directive). The SET directive is useful in establishing temporary or re-usable counters within macros.

4.27 SETDP - SET DIRECT PAGE PSEUDO REGISTER

SETDP <expression> [<comment>]

The SETDP directive is only supported by the M6809 Macro Assembler. It is used to assign a value to the direct page pseudo register at assembly time. The value of the least significant byte of the expression is assigned to the direct page pseudo register. This value is then used in determining if a particular memory reference can use the direct mode of addressing (Paragraph 2.2.4.3). On initialization, the pseudo register is assigned the value zero. Thus, in relocatable programs, direct addressing is automatically generated for BSCT symbols unless the direct page pseudo register has been changed with the SETDP directive. The SETDP directive can be used any number of times in an assembly. Each occurrence changes the value of the direct page pseudo register. The expression cannot contain any external references, forward references, or undefined symbols. In addition, it must be an absolute expression. If the most significant byte of the expression is not zero, a warning occurs. However, the direct page pseudo register is assigned the value of the least significant byte of the expression, anyway.

It should be carefully noted that the SETDP directive does not affect the Direct Page Register at execution time. The user must assume responsibility for ensuring that the assembly and run-time values are compatible. In the example:

SETDP $20

the direct page pseudo register would be set to $20, causing absolute addresses in the range $2000-$20FF to be assembled using the direct addressing mode.

4.28 SPC - SKIP BLANK LINES

SPC <expression>

The SPC directive causes blank lines to be inserted into the source listing for formatting purposes. The SPC directive is not printed in the listing. The number of lines skipped is determined from the expression in the operand field. If the number of lines to be skipped would cause the listing to cross a page boundary, then the paper will only be advanced to the top of the next page. The expression's value must be greater than zero and less than 256. The expression cannot contain any external references or undefined symbols. The value of the expression cannot be relocatable. A source program line that contains only a carriage return will have the same effect in the source listing as the directive "SPC 1".

4.29 TTL - INITIALIZE PAGE HEADING

TTL [<string>]

The TTL directive causes the heading to be initialized to the string in the operand field. Up to 45 printable characters can be specified in the string. If a carriage return is found before the 45th character, the heading will be less than 45 characters. The heading will be printed on the top of all succeeding pages until another TTL directive is encountered. The heading is normally blank except for the Assembler-generated page number.

4.30 XDEF - EXTERNAL SYMBOL DEFINITION

XDEF <symbol>[,<symbol>,...,<symbol>] [<comment>]

The XDEF directive is used to specify that the list of symbols is defined within the current source program, and that those definitions should be passed to the M6800 Linking Loader so that other programs may reference these symbols. This directive is only valid if the program is being assembled with the relocatable option (OPT REL). If the symbols contained in the directive's operand field are not defined in the program, an error will be generated.

4.31 XREF - EXTERNAL SYMBOL REFERENCE

XREF [<sect>:]<sym>[,<sym>,...][,<sect>:<sym>[,<sym>,...]]...

The XREF directive is used to specify that the list of symbols is referenced in the current source program, but is defined (via XDEF directive) in another program. Each <sym> in the operand field of the XREF directive will be associated with a program section, as specified by <sect>. The <sect> specification and the addressing mode assumed for that section can be any one of the following:

<sect>	Addressing mode
BSCT	direct addressing
DSCT	extended addressing
PSCT	extended addressing
ANY	extended addressing

For the M6809 Macro Assembler, direct addressing is only generated for BSCT symbols if the direct page pseudo register is set to zero (Paragraph 4.27). If <sect> is not specified for a symbol, "ANY" will be used as a default. A symbol's section attribute is specified by placing the section name (from above table) followed by a colon (:) in front of the symbol or list of symbols.

If the XREF directive is not used to specify that a symbol is defined in another program, an error will be generated, and all references within the current program to such a symbol will be flagged as undefined.

If, during the subsequent link/load process, the M6800 Linking Loader detects that the section attribute specified for an external reference does not agree with the section attribute of the external definition, an error will be generated. However, this cannot be detected during the assembly process. The use of the ANY section (or no section specification at all) will allow the symbol to be defined in any section.

CHAPTER 5

MACRO OPERATIONS AND CONDITIONAL ASSEMBLY

5.1 INTRODUCTION

This chapter describes the macro and the conditional assembly capabilities of the Macro Assembler. These features can be used in any program, regardless of whether or not the relocation feature is used.

5.2 MACRO OPERATIONS

Programming applications frequently involve the coding of a repeated pattern of instructions that within themselves contain variable entries at each iteration of the pattern, or basic coding patterns subject to conditional assembly at each occurrence may be involved. In either case, macros provide a shorthand notation for handling these patterns. Having determined the iterated pattern, the programmer can, within the macro, designate selectable fields of any statement as variable. Thereafter, by invoking a macro, the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

When the pattern is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently invoked (called). The name of a macro definition should not be the name of an existing instruction mnemonic, a root mnemonic (e.g., SBC, ADD, EOR, etc.), or an Assembler directive.

The macro call causes source statements to be generated. The generated statements may contain substitutable arguments. The statements that may be generated by a macro call are relatively unrestricted as to type. They can be any processor instruction, almost any Assembler directive, or any previously defined macro. Source statements generated by a macro call are subject to the same conditions and restrictions that programmer-generated statements are subject to.

To invoke a macro, the macro name must appear in the operation code field of a source statement. Any arguments are placed into the operand field. By suitably selecting the arguments in relation to their use as indicated by the macro definition, the programmer causes the assembler to produce in-line coding variations of the macro definition.

The effect of a macro call is the same as an open subroutine in that it produces in-line code to perform a predefined function. The in-line code is inserted in the normal flow of the program so that the generated instructions are executed in line with the rest of the program each time the macro is called.

An important feature in defining a macro is the use of macro calls within the macro definition. The Assembler processes such "nested" macro calls at expansion time only. The nesting of one macro definition within another definition, however, is not permitted. If macro names are used as arguments, then they can only be used in the operation field of a macro definition statement if they are to be recognized by the macro processor. Thus, the macro must be defined before its appearance in either a source statement's operation field or in the operand field of another macro call.

In the examples that follow, not all instructions used are recognized by the M6805 Macro Assembler. There is no "B" accumulator, and the "A" accumulator designator is not always required. However, all of the information that follows applies to all Macro Assemblers.

For example, if the following macros were defined in a program:

```
LDAX    MACR
        LDX \0
        LDAA 0,X
        ENDM

LDAXI   MACR
        LDAX \0
        INX
        STX \0
        ENDM
```

then the statement

```
LDAXI   VAR
```

would generate the code

```
LDX     VAR
LDAA    0,X
INX
STX     VAR
```

The definition of macro consists of three parts: the header, which assigns a name to the macro; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the MACR directive and its label. The body contains the pattern of standard source statements. The terminator is the ENDM directive.

For example, if the following code pattern were used in a program:

```
ADDA  LA+5
ADCB  LB+5
SUBA  LC
SBCB  LD
.
.
ADDA  LU
ADCB  LV
SUBA  ALPHA
SBCB  BETA
.
.
ADDA  LW+LX
ADCB  LY+LZ
SUBA  GAMMA
SBCB  DELTA
```

then the following macro definition could be used to represent the above pattern:

```
LDM   MACR
      ADDA   \0
      ADCB   \1
      SUBA   \2
      SBCB   \3
      ENDM
```

Then the previous coding examples could be written using the macro LDM as follows:

```
LDM   LA+5,LB+5,LC,LD
.
.
LDM   LU,LV,ALPHA,BETA
.
.
LDM   LW+LX,LY+LZ,GAMMA,DELTA
```

The Assembler recognizes substitutable arguments by the presence of the backslash character (\). Having encountered this identifier, the Assembler examines the next character which is used as an argument pointer. Argument pointers must be one of the characters in the set of digits 0-9 and the upper case letters A-Z. Thirty-six arguments are the maximum number of arguments that can be handled by any macro definition. Macro arguments can appear anywhere within a source statement of the macro body.

When specifying a symbol in the label field of a statement within the body of a macro, the programmer must be aware that this macro can be used only once, since on the second use, the same label will be redefined, causing an error. Consequently, the user of labels within the macro definition must be approached with caution. Alternatively, the use of Assembler-generated labels, or the placement of substitutable arguments in the label field, is recommended.

The label field, the operation field, and the operand field may all contain text and arguments which can be concatenated by simply placing the substitutable argument directly in the text with no intervening blanks (e.g., AB\0$E). Concatenation is especially useful in the operation field and in the partial subfields of the operand field. As an example, consider a machine instruction such as ADD(R), where (R) can assume the designator A or B. The following macro definition contains a partial operation field argument, as well as a partial operand field:

```
ADJ   MACR
      ADD\0     \1
      AND\0     #\2
      ENDM
```

When the in-line coding is generated, the ADD\0 becomes ADDA or ADDB, as designated by the argument passed along in the macro's argument field. The "AND" instruction is in the immediate mode with the "#" included as part of the macro definition. Thus, the call of the macro ADJ defined above with the following arguments:

```
ADJ    A,TAG1,INDEX
```

would generate the following source statements when expanded:

```
ADDA   TAG1
ANDA   #INDEX
```

Macro usage can be divided into two basic parts: definition and calling (expansion). The definition of macros has been described above. The calling of macros to expand the definition is described below.

The macro call statement is made up of two basic fields: the operation field (contains the macro name) and the operand field (contains substitutable arguments). Each operand of a macro call corresponds one-to-one with an argument pointer of the macro definition. For example, the LDM macro defined earlier could be invoked for expansion (called) by the statement:

```
LDM    LA+5,LB+5,LC,LD
```

where the operand field arguments, separated by commas and taken left to right, correspond to the argument pointers "\0" through "\2", respectively. These arguments are then substituted in their corresponding positions of the definition to produce a sequence of instructions.

The maximum number of macro arguments is 36. These arguments are represented by the argument pointer symbols \0-\9 and \A-\Z in the macro definition. An argument can be declared null when calling a macro. However, it must be declared explicitly null. Null arguments can be specified in two ways: by writing the delimiting commas in succession with no intervening spaces, or by terminating the argument list with a comma and omitting the rest of the argument list. A null argument will cause no character to be substituted in the generated statements that reference the argument. When a macro argument has multiple parts or contains blanks, the argument must be enclosed within parentheses. The parenthetical argument must still be delimited with the normal commas. The parenthetical argument can contain commas as in the following example:

```
LDM    (5,X),(6,X),(LAB+1,X),(LAB+2,X COMMENT)
```

which would generate the following instructions:

```
ADDA    5,X
ADCB    6,X
SUBA    LAB+1,X
SBCB    LAB+2,X    COMMENT
```

It can happen that the argument list of a macro extends beyond the end of a single line. In this case, a semicolon must be used in place of a comma after the last argument to appear on the line. The next argument must then appear in the first column of the next line. This allows for continuation lines. It is illegal to have a semicolon embedded within the text of a parenthetical argument.

At times, labels are required within macros. Since normally a label can only be used once in the label field, multiple macro expansions with the same label will cause multiply defined label errors. One way to avoid this problem is to pass the label to the macro as an argument. Each macro call can then be parameterized with a different label. Another alternative is to use Assembler-generated symbols in the label field. These symbols will take on the form ".nnnnn", where "nnnnn" is a decimal number from 00000 to 65535, inclusive. The Assembler will generate a new symbol whenever it encounters "\.a" within a macro expansion. The "a" must be an alphanumeric character. Each time a new symbol is generated in this manner, an internal counter is incremented. Thus, subsequent symbols encountered in subsequent macro expansions will be unique. Within the same expansion, each reference to the same "\.a" will reference the same symbol generated for that expansion.

The symbol NARG is a special symbol when referenced within a macro expansion. The value assigned to NARG is the number of arguments passed to the current level of macro expansion. This symbol makes it easy to conditionally assemble parts of a macro or to check for error conditions based on the number of passed arguments. Paragraph 5.4 contains several examples of macro usage.

5.3 CONDITIONAL ASSEMBLY

A section of a program that is to be conditionally assembled must be bounded by an IFxx-ENDC directive pair. The source statements following the IFxx directive and up to the next ENDC directive will be included as a part of the source file being assembled only if the condition specified by "xx" is satisfied (true) by the operand field of the IF directive. If the condition proves false, the source file will be assembled as if those statements between the IFxx and the ENDC directives were never encountered.

Conditional assembly allows the user to write a comprehensive source program that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros, and through definition of symbols via the SET and EQU directives. Variations of parameters can then cause assembly of only those parts necessary for the specified conditions.

For instance, a program may be assembled in one of two variations of a basic form, depending on the type of environment in which it will eventually be used. The input/output section of a program, for example, will vary if the program is to be used in a disk environment or in a paper tape environment. Conditional assembly directives can be used to include and to exclude those I/O sections based on a flag set at the beginning of the assembly as shown in the following illustration of a hypothetical program's structure.

```
*
* DEVTYP = 0 MEANS DISK I/O
*     NOT= 0 MEANS TAPE I/O
*

        .
        .
        IFEQ  DEVTYP
        .
        .   DISK I/O SOURCE STATEMENTS
        .
        ENDC
        IFNE  DEVTYP
        .
        .   TAPE I/O SOURCE STATEMENTS
        .
        ENDC
```

When the program above is assembled, one of the I/O sections will be included and one will be excluded from the source file based on the assembly-time value of the symbol "DEVTYP". If the assembly statement:

```
DEVTYP EQU 0
```

is placed into the source file prior to any conditional directive references to that symbol, the disk I/O section will be included and the tape I/O section will be excluded. Similarly, if the statement:

DEVTYP EQU 1

is placed into the source file, the disk I/O section will be excluded and the tape I/O section will be included.

Any of the conditional directives could have been used to effect such a result. Instead of the "equal" and "not equal" conditions, the "greater than" and "less than or equal to" conditions could have been used, etc.

Conditional directives can also be used within a macro definition to ensure at expansion time that the required number of arguments was passed. Specific arguments can be tested to ensure that they fall within a given range of allowable values. In this way, macros can become self-checking and generate error messages to any desired level of detail. The next section contains several examples of conditional assembly directive usage.

5.4 EXAMPLES OF MACROS/CONDITIONAL ASSEMBLY

The following example illustrates the use of a macro and conditional assembly within the macro to check for errors. The macro is used to generate a series of equates for PIA's. The PIA's are assumed to be numbered from 1 to 48 (decimal), inclusive. The addresses of the PIA's start at location $EE00. PIA number 01 occupies locations $EE00-$EE03, PIA number 02 occupies locations $EE04-$EE07, etc. It would be cumbersome to enter all of the equates for all PIA's by hand. Thus, the following macro can be included in a program and invoked to generate those equates required for a given set of PIA's. Error messages are generated via the FAIL directive. The operand field of the FAIL directive is used to identify the error. The example contains sufficient comments to document how the macro works. Following the macro definition are examples of the macro's usage. The example was assembled using the options:

OPT MEX,NOCL

to show the results of the expansion (MEX) and to improve visibility by not printing the conditional directives (NOCL).

```
*
* THE PIA EQUATE MACRO TAKES ONE ARGUMENT.  THE
* ARGUMENT MUST BE A DECIMAL NUMBER BETWEEN 1 AND
* 48, INCLUSIVE.  THE NUMBER MUST BE TWO DIGITS
* (I.E., 01,02,03,...,47,48).
*
* ERRORS WILL BE GENERATED IF ARGUMENT IS MISSING
* IF TOO MANY ARGUMENTS ARE SUPPLIED, OR IF
* THE ARGUMENT IS OUTSIDE OF THE RANGE 01-48.
*
* THE MACRO WILL GENERATE FOUR EQUATES EACH
* TIME IT IS INVOKED.  THE GENERATED EQUATES
* WILL BE FOR THE DATA AND THE CONTROL REGISTERS
* FOR BOTH A AND B SIDES OF THE PIA.
*
PIA    MACR
*
* IF "NARG-1" IS ZERO, ONLY ONE ARGUMENT
* WAS PASSED TO THE MACRO, AS REQUIRED.  IF
* "NARG-1" IS NOT ZERO, TOO FEW OR
* TOO MANY ARGUMENTS WERE PASSED (ERROR).
*
 IFNE NARG-1
  FAIL *TOO FEW OR TOO MANY ARGS*
 ENDC
*
* THE FOLLOWING THREE BLOCKS OF CONDITIONALS
* ARE USED TO CHECK FOR OTHER ERRORS.  THEY WILL
* ONLY BE USED IF THE CORRECT NUMBER OF ARGUMENTS
* WERE PASSED TO THE MACRO (I.E., "NARG-1" = 0).
*
*
* THE NEXT CONDITIONAL TESTS FOR AN ARGUMENT
* VALUE LESS THAN OR EQUAL TO ZERO (INVALID).
* THE "&" IS USED TO FORCE CHECKING FOR A
* VALID, DECIMAL NUMBER
*
 IFEQ NARG-1
  IFLE &\0
   FAIL *PIA <= 0*
  ENDC
 ENDC
*
*
* THE NEXT CONDITIONAL TESTS FOR AN ARGUMENT
* GREATER THAN 48.  IF "\0-49" IS GREATER THAN OR
* EQUAL TO ZERO, THE ARGUMENT WAS GREATER THAN 48
* (INVALID).
*
 IFEQ NARG-1
  IFGE &\0-49
   FAIL *PIA > 48*
  ENDC
```

```
00054                        ENDC
00055                       *
00056                       * THE FOLLOWING CONDITIONALS ARE ONLY TRUE
00057                       * IF NO ERRORS WERE ENCOUNTERED ABOVE.   THE
00058                       * SAME TESTS ARE USED, BUT THE OPPOSITE CONDITION
00059                       * IN ORDER TO REVERSE THE MEANING OF THE TEST.
00060                       *
00061                        IFEQ NARG-1
00062                         IFGT &\0          . ENSURE DECIMAL NUMBER > 0
00063                          IFLT &\0-49      . ENSURE DECIMAL NUMBER < 49
00064                       *
00065                       * GENERATE THE ACTUAL EQUATES
00066                       *
00067                       P\0AD SET $EE00+(\0-1)*4+0 . PIA \0 DATA/DD A
00068                       P\0AC SET $EE00+(\0-1)*4+1 . PIA \0 CONTROL A
00069                       P\0BD SET $EE00+(\0-1)*4+2 . PIA \0 DATA/DD B
00070                       P\0BC SET $EE00+(\0-1)*4+3 . PIA \0 CONTROL B
00071                       *
00072                          ENDC
00073                         ENDC
00074                        ENDC
00075                        ENDM
00076                       *
00077                       * ILLUSTRATE USE OF MACRO TO GENERATE EQUATES
00078                       * FOR PIA NUMBERS 01 AND 04
00079                       *
00080A 0000                       PIA    01
               EE00  A P01AD  SET    $EE00+(01-1)*4+0 . PIA 01 DATA/DD A
               EE01  A P01AC  SET    $EE00+(01-1)*4+1 . PIA 01 CONTROL A
               EE02  A P01BD  SET    $EE00+(01-1)*4+2 . PIA 01 DATA/DD B
               EE03  A P01BC  SET    $EE00+(01-1)*4+3 . PIA 01 CONTROL B
00081A 0000                       PIA    04
               EE0C  A P04AD  SET    $EE00+(04-1)*4+0 . PIA 04 DATA/DD A
               EE0D  A P04AC  SET    $EE00+(04-1)*4+1 . PIA 04 CONTROL A
               EE0E  A P04BD  SET    $EE00+(04-1)*4+2 . PIA 04 DATA/DD B
               EE0F  A P04BC  SET    $EE00+(04-1)*4+3 . PIA 04 CONTROL B
00082                       *
00083                       * THE FOLLOWING USE OF THE MACRO ILLUSTRATES
00084                       * THE ERROR CHECK FOR NO ARGUMENTS PASSED
00085                       *
00086A 0000                       PIA
****ERROR   255--00000
                          FAIL *TOO FEW OR TOO MANY ARGS*
00087                       *
00088                       * THE FOLLOWING USE OF THE MACRO ILLUSTRATES
00089                       * THE ERROR CHECK FOR PIA NUMBER LESS THAN 01
00090                       *
00091A 0000                       PIA    00
****ERROR   255--00086
                           FAIL *PIA <= 0*
00092                       *
00093                       * THE FOLLOWING USE OF THE MACRO ILLUSTRATES
00094                       * THE ERROR CHECK FOR PIA NUMBER GREATER THAN 48
00095                       *
```

```
00096A 0000                  PIA    49
****ERROR   255--00091
                          FAIL *PIA > 48*
00097                  *  ,
00098                  * THE LAST USE OF THE MACRO ILLUSTRATES
00099                  * THE ERROR CHECK FOR TOO MANY ARGUMENTS
00100                  *
00101A 0000                  PIA    01,04
****ERROR   255--00096
                         FAIL *TOO FEW OR TOO MANY ARGS*
00102                        END
TOTAL ERRORS 00004--00101
```

The following example illustrates the use of the Assembler-generated labels within macros. The generated code in itself is meaningless in this example. However, it does validly show how several invocations to the same macro cause different labels to be created.

In this example, no error checking is performed within the macro to ensure that an argument was passed. Thus, if the macro is called without a supplied argument, the "\0" argument pointer will be replaced with a null string (removed) in the generated "JSR" statement. The operand of the JSR will then become the period symbol which was intended to be the first part of a comment. Since "." is a valid Assembler symbol, an undefined symbol error would be generated if the macro were called without an argument.

The following assembly was generated using the options:

OPT MEX,NOCL

to show the results of the expansion (MEX) and to improve visibility by not printing the conditional directives (NOCL).

```
                    *
                    * THE "CALL" MACRO IS USED TO CALL A
                    * SUBROUTINE FOR AN I/O FUNCTION.  PRESUMABLY
                    * THE I/O FUNCTION RETURNS AN ERROR STATUS IN
                    * THE CONDITION CODE REGISTER.  IF THE CARRY
                    * FLAG IS SET TO 1, AN ERROR IS INDICATED.  IF
                    * THE CARRY FLAG IS RESET TO 0, A NORMAL RETURN
                    * IS INDICATED.
                    *
                    * THIS MACRO WILL GENERATE A CALL TO THE FUNCTION
                    * FOLLOWED BY A JUMP INSTRUCTION TO THE ERROR
                    * PROCESSOR.  SINCE THE ERROR PROCESSOR IS
                    * MOST LIKELY OUT OF RANGE FOR A BRANCH
                    * INSTRUCTION, AN UNCONDITIONAL JUMP MUST BE
                    * USED.  THE MACRO WILL AUTOMATICALLY CREATE
                    * INTERMEDIATE LABELS TO BRANCH AROUND THE
                    * JUMP INSTRUCTION.
                    *
                    CALL   MACR
                     JSR \0 . PERFORM I/O
                     BCC \.0 . CC => NO ERROR
                     JMP ERROR . CS => ERROR
                    \.0 EQU * . GENERATED LABEL
                     ENDM
                    *
                    * USING THE "CALL" MACRO
                    *
                    *
                    * DEFINE FICTITIOUS ENTRY POINTS TO THE
                    * INPUT, OUTPUT, AND ERROR ROUTINES.
                    *
            2000  A INPUT  EQU    $2000    . INPUT ROUTINE
            3000  A OUTPUT EQU    $3000    . OUTPUT ROUTINE
            4000  A ERROR  EQU    $4000    . ERROR PROCESSOR
                    *
A 0000                     CALL   INPUT    .
  A 0000 BD 2000  A        JSR    INPUT    . PERFORM I/O
  A 0003 24 03 0008        BCC    .00000   . CC => NO ERROR
  A 0005 7E 4000  A        JMP    ERROR    . CS => ERROR
            0008  A .00000 EQU    *        . GENERATED LABEL
A 0008                     CALL   OUTPUT   .
  A 0008 BD 3000  A        JSR    OUTPUT   . PERFORM I/O
  A 000B 24 03 0010        BCC    .00001   . CC => NO ERROR
  A 000D 7E 4000  A        JMP    ERROR    . CS => ERROR
            0010  A .00001 EQU    *        . GENERATED LABEL
A 0010 20 FE 0010          BRA    *        .
                           END
TOTAL ERRORS 00000--00000
```

The next example utilizes the string forms of the conditional assembly directives (IFC and IFNC). Strings passed as macro arguments tend to be more meaningful than numerical values since they can be descriptive to specify a condition's state. The example could just as well have been written using the value 0 instead of the string "RESET", the value 1 instead of the string "SET", and the value 2 instead of the string "STORE". The comments in the example explain how the macro is used. Following the macro's definition are examples of the macro's usage.

The following example was assembled with the options:

OPT MEX,NOCL

to show the results of the expansion (MEX) and to improve visibility by not printing the conditional directives (NOCL).

```
*
* THE FOLLOWING MACRO ILLUSTRATES THE USE
* OF THE STRING FORM OF THE CONDITIONAL ASSEMBLY
* DIRECTIVES.  AN ARGUMENT IS PASSED TO THE
* MACRO AS A CHARACTER STRING.  BASED ON THE
* VALUE OF THE CHARACTER STRING, THE MACRO
* WILL GENERATE DIFFERENT SEQUENCES OF CODE.
*
* IF THE ARGUMENT "SET" IS SPECIFIED, THE
* INDICATED VARIABLE IN MEMORY WILL BE
* FILLED WITH A PATTERN OF $FF.
* THE VARIABLE NAME IS PASSED AS AN ARGUMENT
* TO THE MACRO ALSO.
*
* IF THE ARGUMENT "RESET" IS SPECIFIED, THE
* INDICATED VARIABLE IN MEMORY WILL BE
* FILLED WITH A PATTERN OF $00.
* THE VARIABLE NAME IS PASSED AS AN ARGUMENT
* TO THE MACRO ALSO.
*
* IF THE ARGUMENT "STORE" IS SPECIFIED, THE
* INDICATED VARIABLE IN MEMORY WILL BE
* FILLED WITH A GIVEN VALUE OR PATTERN.
* THE NAME OF THE VARIABLE AND THE
* VALUE TO BE STORED ARE PASSED AS ARGUMENTS
* TO THE MACRO ALSO.
*
* AN ERROR WILL BE GENERATED IF A STRING
* OTHER THAN "SET", "RESET", OR "STORE" IS
* SPECIFIED AS AN ARGUMENT.
*
BYTE   MACR
*
* CHECK FOR VALID ARGUMENT STRING
*
```

```
00036                       IFNC \0,SET
00037                        IFNC \0,RESET
00038                         IFNC \0,STORE
00039                          FAIL *INVALID STRING ARGUMENT*
00040                         ENDC
00041                        ENDC
00042                       ENDC
00043                      *
00044                      * CHECK FOR "RESET" ARGUMENT
00045                      *
00046                       IFC \0,RESET
00047                       CLR \1 . SET BYTE TO ZERO
00048                       ENDC
00049                      *
00050                      * CHECK FOR "SET" ARUGMENT
00051                      *
00052                       IFC \0,SET
00053                       CLR \1 . SET BYTE TO ZERO
00054                       COM \1 . FLIP TO ALL ONES
00055                       ENDC
00056                      *
00057                      * CHECK FOR "STORE" ARGUMENT
00058                      *
00059                       IFC \0,STORE
00060                       PSHA . SAVE ACCUMULATOR
00061                       LDAA #\1 . GET VALUE
00062                       STAA \2 . STORE VALUE
00063                       PULA . RESTORE ACCUMULATOR
00064                       ENDC
00065                      *
00066                       ENDM
00067                      *
00068                      * USE THE MACRO TO "SET" TEMP1 TO ALL ONES
00069                      *
00070A 0000                      BYTE    SET,TEMP1
     A 0000 7F 0010  A           CLR     TEMP1     . SET BYTE TO ZERO
     A 0003 73 0010  A           COM     TEMP1     . FLIP TO ALL ONES
00071                      *
00072                      * USE THE MACRO TO "RESET" TEMP2 TO ALL ZEROES
00073                      *
00074A 0006                      BYTE    RESET,TEMP2
     A 0006 7F 0011  A           CLR     TEMP2     . SET BYTE TO ZERO
00075                      *
00076                      * USE THE MACRO TO "STORE" ASCII 'A' INTO TEMP3
00077                      *
00078A 0009                      BYTE    STORE,'A,TEMP3
     A 0009 36                   PSHA              . SAVE ACCUMULATOR
     A 000A 86 41    A           LDAA    #'A       . GET VALUE
     A 000C B7 0012  A           STAA    TEMP3     . STORE VALUE
     A 000F 32                   PULA              . RESTORE ACCUMULATOR
00079                      *
00080                      * USE AN INVALID STRING TO SHOW ERROR CHECK
00081                      *
00082A 0010                      BYTE    FILL,A,B,C
```

```
****ERROR   255--00000
                                  FAIL *INVALID STRING ARGUMENT*
00083                       *
00084                       * VARIABLES
00085                       *
00086A 0010      0001   A TEMP1   RMB     1
00087A 0011      0001   A TEMP2   RMB     1
00088A 0012      0001   A TEMP3   RMB     1
00089                             END
TOTAL ERRORS 00001--00082
```

The last example illustrates macro nesting and macro recursion. Nesting refers to calling one macro from within another macro. Recursion refers to calling the same macro from within itself. A recursive macro must have some criterion that can be tested by a conditional assembly directive to prevent infinite recursion. Since macros can only be nested eight levels, the recursive macro can only call itself a maximum of seven times.

The comments in the example will explain how the macro is used. Following the macro definitions are examples of the macro's usage. The example was assembled with the assembly options:

OPT MEX, NOCL

to show the results of the expansion (MEX) and to improve visibility by not printing the conditional directives (NOCL). Within the macro itself the MEX and NOMEX options are used to further clarify the generated expansions by suppressing the printing of the intermediate results of decrementing the recursion counter.

```
00001                       *
00002                       * THE FOLLOWING MACRO CAN BE USED TO REPEAT
00003                       * AN ASSEMBLY LANGUAGE STATEMENT A MAXIMUM
00004                       * OF 42 TIMES.   THE MACRO MAY BE EASILY
00005                       * MODIFIED FOR A LARGER MAXIMUM.
00006                       *
00007                       * THE REPEAT MACRO INVOKES  ANOTHER MACRO WHICH
00008                       * IS USED TO REPEAT THE ASSEMBLY STATEMENT
00009                       * A MAXIMUM OF 7 TIMES (HENCE THE NAME
00010                       * RPT1_7).   THE REPEAT MACRO MAINTAINS A COUNTER
00011                       * WHICH IS DECREMENTED BY 7 EACH TIME THE
00012                       * INNER MACRO IS CALLED.   WHEN THE COUNTER
00013                       * HAS A VALUE OF 7 OR LESS, THE
00014                       * INNER MACRO IS CALLED ONE FINAL TIME TO FINISH
00015                       * THE REPETITION OF THE REMAINING LINES.
00016                       *
00017                       REPEAT MACR
00018                        OPT NOMEX
00019                        IFGT \0-42
00020                         FAIL * COUNT EXCEEDS MAXIMUM *
```

```
                 ENDC
                .COUNT SET \0 .  INITIAL VALUE OF COUNTER
                 IFGE .COUNT-7
                .COUNT SET .COUNT-7 .  REDUCE BY SEVEN
                 RPT1_7 7,(\1) .  DO 7 LINES
                 ENDC
                *
                 IFGE .COUNT-7
                .COUNT SET .COUNT-7 .  REDUCE BY SEVEN MORE (14)
                 RPT1_7 7,(\1) .  DO 7 LINES
                 ENDC
                *
                 IFGE .COUNT-7
                .COUNT SET .COUNT-7 .  REDUCE BY SEVEN MORE (21)
                 RPT1_7 7,(\1) .  DO 7 LINES
                 ENDC
                *
                 IFGE .COUNT-7
                .COUNT SET .COUNT-7 .  REDUCE BY SEVEN MORE (28)
                 RPT1_7 7,(\1) .  DO 7 LINES
                 ENDC
                *
                 IFGE .COUNT-7
                .COUNT SET .COUNT-7 .  REDUCE BY SEVEN MORE (35)
                 RPT1_7 7,(\1) .  DO 7 LINES
                 ENDC
                *
                 RPT1_7 .COUNT,(\1) DO REMAINING LINES
                 OPT MEX
                 ENDM
                *
                * INNER MACRO--RECURSIVE
                *
                RPT1_7 MACR
                .T SET \0
                 OPT MEX
                 \1
                 OPT NOMEX
                .T SET .T-1
                 IFNE .T
                 RPT1_7 .T,(\1)
                 ENDC
                 ENDM
                *
                * USE MACRO TO GENERATE TABLE OF THE POWERS OF
                * TWO.  THE TABLE CAN BE LOCATED ANYWHERE
                * SINCE THE EXPRESSION SUBTRACTS THE PROGRAM
                * COUNTER FROM THE BASE ADDRESS OF THE TABLE.
                *
A ABCD                  ORG     $ABCD       ILLUSTRATE INDEPENDENCE
                *
A ABCD          BASE    REPEAT 16,(FDB  2!^((*-BASE)/2))
                        OPT     MEX
A ABCD    0001  A       FDB     2!^((*-BASE)/2)
```

```
                                OPT    MEX
     A ABCF    0002  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABD1    0004  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABD3    0008  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABD5    0010  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABD7    0020  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABD9    0040  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABDB    0080  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABDD    0100  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABDF    0200  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABE1    0400  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABE3    0800  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABE5    1000  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABE7    2000  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABE9    4000  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
     A ABEB    8000  A          FDB    2!^((*-BASE)/2)
                                OPT    MEX
00073                      *
00074                      * USE MACRO TO GENERATE VARIABLE NUMBER OF
00075                      * SHIFT INSTRUCTIONS
00076                      *
00077          0005  A V1       EQU    5
00078A ABED                     REPEAT V1,(ASRA)
                                OPT    MEX
     A ABED 47                  ASRA
                                OPT    MEX
     A ABEE 47                  ASRA
                                OPT    MEX
     A ABEF 47                  ASRA
                                OPT    MEX
     A ABF0 47                  ASRA
                                OPT    MEX
     A ABF1 47                  ASRA
                                OPT    MEX
00079                           END
TOTAL ERRORS 00000--00000
```

APPENDIX A

CHARACTER SET

The character set recognized by the Macro Assembler 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.
2. The digits 0 through 9.
3. Four arithmetic operators: +, -, *, and /.
4. The special two-character operators: !^, !>, !<, !X, !., !+, !R, and !L.
5. Parentheses in expression: (,).
6. The special symbol characters: underscore (_), period (.), and dollar sign ($). Only the period may be used as the first character of a symbol.
7. The characters used as prefixes for constants and addressing modes:

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

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

,X	Indexed addressing
H	Hexadecimal constant
O	Octal constant
Q	Octal constant
B	Binary constant
,PCR	M6809 indexed addressing
,S	M6809 indexed addressing
,U	M6809 indexed addressing
,Y	M6809 indexed addressing

9. Three separator characters: space, carriage return, and comma.
10. The character "*" to indicate comments. Comments may contain any printable characters from the ASCII set.
11. The special symbols "\" and "\." used with the macro definitions as argument pointers or Assembler-generated symbols, respectively.
12. For the M6800/M6801 and M6809 Macro Assemblers, the special symbols "A" and "B" to specify the accumulator in the operation code. For the M6805 Macro Assembler, the special symbols "A" and "X" to specify the accumulator or index register in the operation code. The special symbol "X" to indicate indexed addressing in the operand field; the special symbol "*" to represent the value of the current program counter; and the special symbol "NARG" to represent the number of macro arguments passed to the current level of macro expansion. For

the M6809 Macro Assembler, the special symbols "PCR", "S", "U", and "Y" to indicate indexed addressing in the operand field; the special symbol "D" to specify the accumulator in the operation code; the special symbols "A", "B", "CC", "D", "DP", "PC", "S", "U", "X", and "Y" to indicate registers in the operand field of the TFR, EXG, PSHU, PULU, PSHS, and PULS instructions; and the special symbols "A", "B", and "D" to indicate offsets in the indexed mode.

13. For the M6809 Macro Assembler, the characters used to indicate indirect addressing: [,].

14. For the M6809 Macro 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.

15. For the M6809 Macro 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
0	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

SUMMARY OF INSTRUCTIONS

The following table lists the special symbols used in the description of M6800, M6801, M6805, and M6809 instructions.

Operation Functions

=	Left side of equal sign is replaced by right side of equal sign
[]	Evaluate contents first; grouping
()	The contents of
M()	The contents of memory specified by the parenthetical address
+	Arithmetic addition
-	Arithmetic subtraction
*	Arithmetic multiplication
and	Boolean and
effad	M6809 effective address
or	Boolean inclusive or
xor	Boolean exclusive or
L>	Logical shift right by number of bits specified
L<	Logical shift left by number of bits specified
A>	Arithmetic shift right by number of bits specified
A<	Arithmetic shift left by number of bits specified
R>	Rotate right by number of bits specified
R<	Rotate left by number of bits specified

Operand Sizes and Register Names

$nn	The hexadecimal number "nn"
n	A bit value of n (0 or 1)
nn	An eight-bit value of nn (00-$FF)
nnnn	A sixteen-bit value of nnnn (0000-$FFFF)
aa	Eight-bit address
aaaa	Sixteen-bit address
A	Accumulator A
B	M6800/M6801/M6809 Accumulator B
C	Carry condition code (Bit 0 of CC)
CC	Condition code register
D	M6801/M6809 dual accumulator A,B
EI	M6805 external interrupt pin
F	M6809 fast interrupt condition code (Bit 6 of CC)
H	Half carry condition code (Bit 5 of CC; bit 4 if M6805)
I	Interrupt condition code (Bit 4 of CC; bit 3 if M6805)
ii	Eight-bit immediate operand
iiii	Sixteen-bit immediate operand
N	Sign condition code (Bit 3 of CC; bit 2 if M6805)
P	Program counter register
rl	M6809 register list
rr	Eight-bit, relative branch address
rrrr	Sixteen bit, relative branch address

S	Stack register
U	M6809 user stack register
V	M6800/M6801/M6809 overflow condition code (Bit 1 of CC)
X	Index register
xx	Eight-bit, indexed addressing offset
xxop	M6809 indexed operation depends on index mode (see B.5)
xx0	M6805 no offset indexed addressing
xx1	M6805 eight-bit, indexed addressing offset
xx2	M6805 sixteen-bit, indexed addressing offset
Y	M6809 index register
Z	Zero condition code (Bit 2 of CC; bit 1 if M6805)

Condition code symbols

T	Status bit tested and set if true; reset otherwise
0	Status bit reset by operation
1	Status bit set by operation
-	Status bit unaffected by operation
?	Programming Reference Manual contains details on setting of the status bit

B.1 M6800 INSTRUCTIONS

In the following tables, the "Function" column for branch instructions only contains the test condition performed by the branch. The following function will be performed if the result of the test is true:

P=(P)+0002+rr

If the result of the test is false, the following function will be performed:

P=(P)+0002

The functions for the instructions BSR, DAA, JSR, RTI, RTS, SWI, and WAI are described in detail in the M6800 Programming Reference Manual.

Mne- monic	Oper- and	Op- code	Function	H	I	N	Z	V	C
ABA	--	1B	A=(A)+(B)	T	-	T	T	T	T
ADCA	ii	89	A=(A)+ii+(C)	T	-	T	T	T	T
	aa	99	A=(A)+ M(aa)+(C)						
	xx	A9	A=(A)+M((X)+xx)+(C)						
	aaaa	B9	A=(A)+M(aaaa)+(C)						
ADCB	ii	C9	B=(B)+ii+(C)	T	-	T	T	T	T
	aa	D9	B=(B)+M(aa)+(C)						
	xx	E9	B=(B)+M((X)+xx)+(C)						
	aaaa	F9	B=(B)+M(aaaa)+(C)						
ADDA	ii	8B	A=(A)+ii	T	-	T	T	T	T
	aa	9B	A=(A)+M(aa)						
	xx	AB	A=(A)+M((X)+xx)						
	aaaa	BB	A=(A)+M(aaaa)						
ADDB	ii	CB	B=(B)+ii	T	-	T	T	T	T
	aa	DB	B=(B)+M(aa)						
	xx	EB	B=(B)+M((X)+xx)						
	aaaa	FB	B=(B)+M(aaaa)						
ANDA	ii	84	A=(A) and ii	-	-	T	T	0	-
	aa	94	A=(A) and M(aa)						
	xx	A4	A=(A) and M((X)+xx)						
	aaaa	B4	A=(A) and M(aaaa)						
ANDB	ii	C4	B=(B) and ii	-	-	T	T	0	-
	aa	D4	B=(B) and M(aa)						
	xx	E4	B=(B) and M((X)+xx)						
	aaaa	F4	B=(B) and M(aaaa)						
ASL	xx	68	M((X)+xx)=M((X)+xx) A< 1	-	-	T	T	?	T
	aaaa	78	M(aaaa)=M(aaaa) A< 1						
ASLA	--	48	A=(A) A< 1	-	-	T	T	?	T
ASLB	--	58	B=(B) A< 1	-	-	T	T	?	T
ASR	xx	67	M((X)+xx)=M((X)+xx) A> 1	-	-	T	T	?	T
	aaaa	77	M(aaaa)=M(aaaa) A> 1						
ASRA	--	47	A=(A) A> 1	-	-	T	T	?	T
ASRB	--	57	B=(B) A> 1	-	-	T	T	?	T
BCC	rr	24	Test (C)=0	-	-	-	-	-	-
BCS	rr	25	Test (C)=1	-	-	-	-	-	-
BEQ	rr	27	Test (Z)=1	-	-	-	-	-	-
BGE	rr	2C	Test (N) xor (V)=0	-	-	-	-	-	-
BGT	rr	2E	Test (Z) or [(N) xor (V)]=0	-	-	-	-	-	-
BHI	rr	22	Test (C) xor (Z)=0	-	-	-	-	-	-
BITA	ii	85	(A) and ii	-	-	T	T	0	-
	aa	95	(A) and M(aa)						
	xx	A5	(A) and M((X)+xx)						
	aaaa	B5	(A) and M(aaaa)						
BITB	ii	C5	(B) and ii	-	-	T	T	0	-
	aa	D5	(B) and M(aa)						
	xx	E5	(B) and M((X)+xx)						
	aaaa	F5	(B) and M(aaaa)						
BLE	rr	2F	Test (Z) or [(N) xor (V)]=1	-	-	-	-	-	-
BLS	rr	23	Test (C) or (Z)=1	-	-	-	-	-	-
BLT	rr	2D	Test (N) xor (V)=1	-	-	-	-	-	-

Mne-	Oper-	Op-	Function	Status					
monic	and	code		H	I	N	Z	V	C
BMI	rr	2B	Test (N)=1	-	-	-	-	-	-
BNE	rr	26	Test (Z)=0	-	-	-	-	-	-
BPL	rr	2A	Test (N)=0	-	-	-	-	-	-
BRA	rr	20	Tests always true	-	-	-	-	-	-
BSR	rr	8D	Subroutine call	-	-	-	-	-	-
BVC	rr	28	Test (V)=0	-	-	-	-	-	-
BVS	rr	29	Test (V)=1	-	-	-	-	-	-
CBA	--	11	(A)-(B)	-	-	T	T	T	T
CLC	--	0C	C=0	-	-	-	-	-	0
CLI	--	0E	I=0	-	0	-	-	-	-
CLR	xx	6F	M((X)+xx)=00	-	-	0	1	0	0
	aaaa	7F	M(aaaa)=00						
CLRA	--	4F	A=00	-	-	0	1	0	0
CLRB	--	5F	B=00	-	-	0	1	0	0
CLV	--	0A	V=0	-	-	-	-	0	-
CMPA	ii	81	(A)-ii	-	-	T	T	T	T
	aa	91	(A)-M(aa)						
	xx	A1	(A)-M((X)+xx)						
	aaaa	B1	(A)-M(aaaa)						
CMPB	ii	C1	(B)-ii	-	-	T	T	T	T
	aa	D1	(B)-M(aa)						
	xx	E1	(B)-M((X)+xx)						
	aaaa	F1	(B)-M(aaaa)						
COM	xx	63	M((X)+xx)=M((X)+xx) xor $FF	-	-	T	T	0	1
	aaaa	73	M(aaaa)=M(aaaa) xor $FF						
COMA	--	43	A=(A) xor $FF	-	-	T	T	0	1
COMB	--	53	B=(B) xor $FF	-	-	T	T	0	1
CPX	iiii	8C	(X)-iiii	-	-	?	T	?	-
	aa	9C	(X)-M(aa,aa+1)						
	xx	AC	(X)-M((X)+xx,(X)+xx+1)						
	aaaa	BC	(X)-M(aaaa,aaaa+1)						
DAA	--	19	Converts binary add of BCD into BCD	-	-	T	T	T	?
DEC	xx	6A	M((X)+xx)=M((X)+xx)-01	-	-	T	T	?	-
	aaaa	7A	M(aaaa)=M(aaaa)-01						
DECA	--	4A	A=(A)-01	-	-	T	T	?	-
DECB	--	5A	B=(B)-01	-	-	T	T	?	-
DES	--	34	S=(S)-0001	-	-	-	-	-	-
DEX	--	09	X=(X)-0001	-	-	-	T	-	-
EORA	ii	88	A=(A) xor ii	-	-	T	T	0	-
	aa	98	A=(A) xor M(aa)						
	xx	A8	A=(A) xor M((X)+xx)						
	aaaa	B8	A=(A) xor M(aaaa)						
EORB	ii	C8	B=(B) xor ii	-	-	T	T	0	-
	aa	D8	B=(B) xor M(aa)						
	xx	E8	B=(B) xor M((X)+xx)						
	aaaa	F8	B=(B) xor M(aaaa)						
INC	xx	6C	M((X)+xx)=M((X)+xx)+01	-	-	T	T	?	-
	aaaa	7C	M(aaaa)=M(aaaa)+01						
INCA	--	4C	A=(A)+01	-	-	T	T	?	-

Mne-monic	Oper-and	Op-code	Function	H	I	N	Z	V	C
INCB	--	5C	B=(B)+01	-	-	T	T	?	-
INS	--	31	S=(S)+0001	-	-	-	-	-	-
INX	--	08	X=(X)+0001	-	-	-	T	-	-
JMP	xx	6E	P=(X)+xx	-	-	-	-	-	-
	aaaa	7E	P=aaaa						
JSR	xx	AD	Subroutine call	-	-	-	-	-	-
	aaaa	BD	Subroutine call						
LDAA	ii	86	A=ii	-	-	T	T	0	-
	aa	96	A=M(aa)						
	xx	A6	A=M((X)+xx)						
	aaaa	B6	A=M(aaaa)						
LDAB	ii	C6	B=ii	-	-	T	T	0	-
	aa	D6	B=M(aa)						
	xx	E6	B=M((X)+xx)						
	aaaa	F6	B=M(aaaa)						
LDS	iiii	8E	S=iiii	-	-	?	T	0	-
	aa	9E	S=M(aa,aa+1)						
	xx	AE	S=M((X)+xx,(X)+xx+1)						
	aaaa	BE	S=M(aaaa,aaaa+1)						
LDX	iiii	CE	X=iiii	-	-	?	T	0	-
	aa	DE	X=M(aa,aa+1)						
	xx	EE	X=M((X)+xx,(X)+xx+1)						
	aaaa	FE	X=M(aaaa,aaaa+1)						
LSR	xx	64	M((X)+xx)=M((X)+xx) L> 1	-	-	0	T	?	T
	aaaa	74	M(aaaa)=M(aaaa) L> 1						
LSRA	--	44	A=(A) L> 1	-	-	0	T	?	T
LSRB	--	54	B=(B) L> 1	-	-	0	T	?	T
NEG	xx	60	M((X)+xx)=00-M((X)+xx)	-	-	T	T	?	?
	aaaa	70	M(aaaa)=00-M(aaaa)						
NEGA	--	40	A=00-(A)	-	-	T	T	?	?
NEGB	--	50	B=00-(B)	-	-	T	T	?	?
NOP	--	01	P=(P)+0001	-	-	-	-	-	-
ORAA	ii	8A	A=(A) or ii	-	-	T	T	0	-
	aa	9A	A=(A) or M(aa)						
	xx	AA	A=(A) or M((X)+xx)						
	aaaa	BA	A=(A) or M(aaaa)						
ORAB	ii	CA	B=(B) or ii	-	-	T	T	0	-
	aa	DA	B=(B) or M(aa)						
	xx	EA	B=(B) or M((X)+xx)						
	aaaa	FA	B=(B) or M(aaaa)						
PSHA	--	36	M(S)=A; S=(S)-0001	-	-	-	-	-	-
PSHB	--	37	M(S)=B; S=(S)-0001	-	-	-	-	-	-
PULA	--	32	S=(S)+0001; A=M(S)	-	-	-	-	-	-
PULB	--	33	S=(S)+0001; B=M(S)	-	-	-	-	-	-
ROL	xx	69	M((X)+xx)=M((X)+xx) R< 1	-	-	T	T	?	T
	aaaa	79	M(aaaa)=M(aaaa) R< 1						
ROLA	--	49	A=(A) R< 1	-	-	T	T	?	T
ROLB	--	59	B=(B) R< 1	-	-	T	T	?	T
ROR	xx	66	M((X)+xx)=M((X)+xx) R> 1	-	-	T	T	?	T
	aaaa	76	M(aaaa)=M(aaaa) R> 1						

Mne-monic	Oper-and	Op-code	Function	H	I	N	Z	V	C
RORA	--	46	A=(A) R> 1	-	-	T	T	?	T
RORB	--	56	B=(B) R> 1	-	-	T	T	?	T
RTI	--	3B	Return from interrupt	?	?	?	?	?	?
RTS	--	39	Return from subroutine	-	-	-	-	-	-
SBA	--	10	A=(A)-(B)	-	-	T	T	T	T
SBCA	ii	82	A=(A)-ii-(C)	-	-	T	T	T	T
	aa	92	A=(A)-M(aa)-(C)						
	xx	A2	A=(A)-M((X)+xx)-(C)						
	aaaa	B2	A=(A)-M(aaaa)-(C)						
SBCB	ii	C2	B=(B)-ii-(C)	-	-	T	T	T	T
	aa	D2	B=(B)-M(aa)-(C)						
	xx	E2	B=(B)-M((X)+xx)-(C)						
	aaaa	F2	B=(B)-M(aaaa)-(C)						
SEC	--	0D	C=1	-	-	-	-	-	1
SEI	--	0F	I=1	-	1	-	-	-	-
SEV	--	0B	V=1	-	-	-	-	1	-
STAA	aa	97	M(aa)=(A)	-	-	T	T	0	-
	xx	A7	M((X)+xx)=(A)						
	aaaa	B7	M(aaaa)=(A)						
STAB	aa	D7	M(aa)=(B)	-	-	T	T	0	-
	xx	E7	M((X)+xx)=(B)						
	aaaa	F7	M(aaaa)=(B)						
STS	aa	9F	M(aa,aa+1)=(S)	-	-	?	T	0	-
	xx	AF	M((X)+xx,(X)+xx+1)=(S)						
	aaaa	BF	M(aaaa,aaaa+1)=(S)						
STX	aa	DF	M(aa,aa+1)=(X)	-	-	?	T	0	
	xx	EF	M((X)+xx,(X)+xx+1)=(X)						
	aaaa	FF	M(aaaa,aaaa+1)=(X)						
SUBA	ii	80	A=(A)-ii	-	-	T	T	T	T
	aa	90	A=(A)-M(aa)						
	xx	A0	A=(A)-M((X)+xx)						
	aaaa	B0	A=(A)-M(aaaa)						
SUBB	ii	C0	B=(B)-ii	-	-	T	T	T	T
	aa	D0	B=(B)-M(aa)						
	xx	E0	B=(B)-M((X)+xx)						
	aaaa	F0	B=(B)-M(aaaa)						
SWI	--	3F	Software interrupt	-	1	-	-	-	- -
TAB	--	16	B=(A)	-	-	T	T	0	-
TAP	--	06	CC=(A)	?	?	?	?	?	?
TBA	- -	17	A=(B)	-	-	T	T	0	-
TPA	--	07	A=(CC)	-	-	-	-	-	-
TST	xx	6D	M((X)+xx)-00	-	-	T	T	0	0
	aaaa	7D	M(aaaa)-00						
TSTA	--	4D	(A)-00	-	-	T	T	0	0
TSTB	--	5D	(B)-00	-	-	T	T	0	0
TSX	--	30	X=(S)+0001	-	-	-	-	-	-
TXS	--	35	S=(X)-0001	-	-	-	-	-	-
WAI	--	3E	Wait for IRQ	-	?	-	-	-	-

B.2 M6801 INSTRUCTIONS

The M6801 allows all of the instructions from the preceding table. In addition, the following instructions are valid. These instructions can only be assembled using the MDOS or tape version of the M6800 Macro Assembler.

Mnemonic	Operand	Opcode	Function	H	I	N	Z	V	C
ABX	--	3A	X=(X)+(B)	-	-	-	-	-	-
ADDD	iiii	C3	D=(D)+iiii	-	-	T	T	T	T
	aa	D3	D=(D)+M(aa,aa+1)						
	xx	E3	D=(D)+M((X)+xx,(X)+xx+1)						
	aaaa	F3	D=(D)+M(aaaa,aaaa+1)						
ASLD	--	05	D=(D) A< 1	-	-	T	T	?	T
BHS	rr	24	Test (C)=0	-	-	-	-	-	-
BLO	rr	25	Test (C)=1	-	-	-	-	-	-
BRN	rr	21	Tests always false	-	-	-	-	-	-
JSR	aa	9D	Subroutine call	-	-	-	-	-	-
LDD	iiii	CC	D=iiii	-	-	T	T	0	-
	aa	DC	D=M(aa,aa+1)						
	xx	EC	D=M((X)+xx,(X)+xx+1)						
	aaaa	FC	D=M(aaaa,aaaa+1)						
LSL	xx	68	M((X)+xx)=M((X)+xx) L< 1	-	-	T	T	?	T
	aaaa	78	M(aaaa)=M(aaaa) L< 1						
LSLA	--	48	A=(A) L< 1	-	-	T	T	?	T
LSLB	--	58	B=(B) L< 1	-	-	T	T	?	T
LSLD	--	05	D=(D) A< 1	-	-	T	T	?	T
LSRD	--	04	D=(D) L> 1	-	-	0	T	?	T
MUL	--	3D	D=(A)*(B)	-	-	-	-	-	?
PSHX	--	3C	M(S,S+1)=(X); S=(S)-0002	-	-	-	-	-	-
PULX	--	38	S=(S)+0002; X=M(S,S+1)	-	-	-	-	-	-
STD	aa	DD	M(aa,aa+1)=(D)	-	-	T	T	0	-
	xx	ED	M((X)+xx,(X)+xx+1)=(D)						
	aaaa	FD	M(aaaa,aaaa+1)=(D)						
SUBD	iiii	83	D=(D)-iiii	-	-	T	T	T	T
	aa	93	D=(D)-M(aa,aa+1)						
	xx	A3	D=(D)-M((X)+xx,(X)+xx+1)						
	aaaa	B3	D=(D)-M(aaaa,aaaa+1)						

B.3 M6805 INSTRUCTIONS

In the following tables, the "Function" column for branch instructions only contains the test condition performed by the branch. The following function will be performed if the result of the test is true:

P=(P)+0002+rr (for branch)

P=(P)+0003+rr (for bit test and branch)

If the result of the test is false, the following function will be performed:

P=(P)+0002 (for branch)

P=(P)+0003 (for bit test and branch)

The functions for the instructions BSR, JSR, RTI, RTS, STOP, SWI, and WAIT are described in detail in the M6805 Programming Reference Manual.

Mnemonic	Operand	Opcode	Function	H	I	N	Z	C
ADC	ii	A9	A=(A)+ii+(C)	T	-	T	T	T
	aa	B9	A=(A)+M(aa)+(C)					
	aaaa	C9	A=(A)+M(aaaa)+(C)					
	xx2	D9	A=(A)+M((X)+xx2)+(C)					
	xx1	E9	A=(A)+M((X)+xx1)+(C)					
	xx0	F9	A=(A)+M(X)+(C)					
ADD	ii	AB	A=(A)+ii	T	-	T	T	T
	aa	BB	A=(A)+M(aa)					
	aaaa	CB	A=(A)+M(aaaa)					
	xx2	DB	A=(A)+M((X)+xx2)					
	xx1	EB	A=(A)+M((X)+xx1)					
	xx0	FB	A=(A)+M(X)					
AND	ii	A4	A=(A) and ii	-	-	T	T	-
	aa	B4	A=(A) and M(aa)					
	aaaa	C4	A=(A) and M(aaaa)					
	xx2	D4	A=(A) and M((X)+xx2)					
	xx1	E4	A=(A) and M((X)+xx1)					
	xx0	F4	A=(A) and M(X)					
ASL	aa	38	M(aa)=M(aa) A< 1	-	-	T	T	T
	xx1	68	M((X)+xx1)=M((X) A< 1					
	xx0	78	M(X)=M(X) A< 1					
ASLA	--	48	A=(A) A< 1	-	-	T	T	T
ASLX	--	58	X=(X) A< 1	-	-	T	T	T
ASR	aa	37	M(aa)=M(aa) A> 1	-	-	T	T	T
	xx1	67	M((X)+xx1)=M((X)+xx1) A> 1					
	xx0	77	M(X)=M(X) A> 1					
ASRA	--	47	A=(A) A> 1	-	-	T	T	T
ASRX	--	57	X=(X) A> 1	-	-	T	T	T
BCC	rr	24	Test (C)=0	-	-	-	-	-
BCLR	0,aa	11	Bit 0 of M(aa)=0	-	-	-	-	-
	1,aa	13	Bit 1 of M(aa)=0					
	2,aa	15	Bit 2 of M(aa)=0					
	3,aa	17	Bit 3 of M(aa)=0					
	4,aa	19	Bit 4 of M(aa)=0					
	5,aa	1B	Bit 5 of M(aa)=0					
	6,aa	1D	Bit 6 of M(aa)=0					
	7,aa	1F	Bit 7 of M(aa)=0					
BCS	rr	25	Test (C)=1	-	-	-	-	-
BEQ	rr	27	Test (Z)=1	-	-	-	-	-
BHCC	rr	28	Test (H)=0	-	-	-	-	-
BHCS	rr	29	Test (H)=1	-	-	-	-	-
BHI	rr	22	Test (C) xor (Z)=0	-	-	-	-	-
BHS	rr	24	Test (C)=0	-	-	-	-	-
BIH	rr	2F	Test EI =high	-	-	-	-	-
BIL	rr	2E	Test EI =low	-	-	-	-	-

Mne-monic	Oper-and	Op-code	Function	Status H I N Z C
BIT	ii	A5	(A) and ii	- - T T -
	aa	B5	(A) and M(aa)	
	aaaa	C5	(A) and M(aaaa)	
	xx2	D5	(A) and M((X)+xx2)	
	xx1	E5	(A) and M((X)+xx1)	
	xx0	F5	(A) and M(X)	
BLO	rr	25	Test (C)=1	- - - - -
BLS	rr	23	Test (C) or (Z)=1	- - - - -
BMC	rr	2C	Test (I)=0	- - - - -
BMI	rr	2B	Test (N)=1	- - - - -
BMS	rr	2D	Test (I)=1	- - - - -
BNE	rr	26	Test (Z)=0	- - - - -
BPL	rr	2A	Test (N)=0	- - - - -
BRA	rr	20	Tests always true	- - - - -
BRCLR	0,aa,rr	01	Test bit 0 of M(aa)=0	- - - - T
	1,aa,rr	03	Test bit 1 of M(aa)=0	
	2,aa,rr	05	Test bit 2 of M(aa)=0	
	3,aa,rr	07	Test bit 3 of M(aa)=0	
	4,aa,rr	09	Test bit 4 of M(aa)=0	
	5,aa,rr	0B	Test bit 5 of M(aa)=0	
	6,aa,rr	0D	Test bit 6 of M(aa)=0	
	7,aa,rr	0F	Test bit 7 of M(aa)=0	
BRN	rr	21	Tests always false	- - - - -
BRSET	0,aa,rr	00	Test bit 0 of M(aa)=1	- - - - -
	1,aa,rr	02	Test bit 1 of M(aa)=1	
	2,aa,rr	04	Test bit 2 of M(aa)=1	
	3,aa,rr	06	Test bit 3 of M(aa)=1	
	4,aa,rr	08	Test bit 4 of M(aa)=1	
	5,aa,rr	0A	Test bit 5 of M(aa)=1	
	6,aa,rr	0C	Test bit 6 of M(aa)=1	
	7,aa,rr	0E	Test bit 7 of M(aa)=1	
BSET	0,aa	10	Bit 0 of M(aa)=1	- - - - -
	1,aa	12	Bit 1 of M(aa)=1	
	2,aa	14	Bit 2 of M(aa)=1	
	3,aa	16	Bit 3 of M(aa)=1	
	4,aa	18	Bit 4 of M(aa)=1	
	5,aa	1A	Bit 5 of M(aa)=1	
	6,aa	1C	Bit 6 of M(aa)=1	
	7,aa	1E	Bit 7 of M(aa)=1	
BSR	rr	AD	Subroutine call	- - - - -
CLC	---	98	C=0	- - - - 0
CLI	---	9A	I=0	- 0 - - -
CLR	aa	3F	M(aa)=00	- - 0 1 -
	xx1	6F	M((X)+xx1)=00	
	xx0	7F	M(X)=00	
CLRA	---	4F	A=00	- - 0 1 -
CLRX	---	5F	X=00	- - 0 1 -

Mnemonic	Operand	Opcode	Function	H	I	N	Z	C
CMP/	ii	A1	(A)-ii	-	-	T	T	T
CMPA	aa	B1	(A)-M(aa)					
	aaaa	C1	(A)-M(aaaa)					
	xx2	D1	(A)-M((X)+xx2)					
	xx1	E1	(A)-M((X)+xx1)					
	xx0	F1	(A)-M(X)					
COM	aa	33	M(aa)=M(aa) xor $FF	-	-	T	T	1
	xx1	63	M((X)+xx1)=M((X)+xx1) xor $FF					
	xx0	73	M(X)= M(X) xor $FF					
COMA	---	43	A=(A) xor $FF	-	-	T	T	1
COMX	---	53	X=(X) xor $FF	-	-	T	T	1
CPX/	ii	A3	(X)-ii	-	-	T	T	T
CMPX	aa	B3	(X)-M(aa)					
	aaaa	C3	(X)-M(aaaa)					
	xx2	D3	(X)-M((X)+xx2)					
	xx1	E3	(X)-M((X)+xx1)					
	xx0	F3	(X)-M(X)					
DEC	aa	3A	M(aa)=M(aa)-01	-	-	T	T	-
	xx1	6A	M((X)+xx1)=M((X)+xx1)-01					
	xx0	7A	M(X)=M(X)-01					
DECA	---	4A	A=(A)-01	-	-	T	T	-
DECX/ DEX	---	5A	X=(X)-01	-	-	T	T	- -
EOR	ii	A8	A=(A) xor ii	-	-	T	T	-
	aa	B8	A=(A) xor M(aa)					
	aaaa	C8	A=(A) xor M(aaaa)					
	xx2	D8	A=(A) xor M((X)+xx2)					
	xx1	E8	A=(A) xor M((X)+xx1)					
	xx0	F8	A=(A) xor M(X)					
INC	aa	3C	M(aa)=M(aa)+01	-	-	T	T	-
	xx1	6C	M((X)+xx1)=M((X)+xx1)+01					
	xx0	7C	M(X)=M(X)+01					
INCA	---	4C	A=(A)+01	-	-	T	T	-
INCX/ INX	---	5C	X=(X)+01	-	-	T	T	-
JMP	aa	BC	P=aa	-	-	-	-	-
	aaaa	CC	P=aaaa					
	xx2	DC	P=(X)+xx2					
	xx1	EC	P=(X)+xx1					
	xx0	FC	P=(X)					
JSR	aa	BD	Subroutine call	-	-	-	-	-
	aaaa	CD	Subroutine call					
	xx2	DD	Subroutine call					
	xx1	ED	Subroutine call					
	xx0	FD	Subroutine call					

Mnemonic	Operand	Opcode	Function	H	I	N	Z	C
LDA	ii	A6	A=ii	-	-	T	T	-
	aa	B6	A=M(aa)					
	aaaa	C6	A=M(aaaa)					
	xx2	D6	A=M((X)+xx2)					
	xx1	E6	A=M((X)+xx1)					
	xx0	F6	A=M(X)					
LDX	ii	AE	X=ii	-	-	T	T	-
	aa	BE	X=M(aa)					
	aaaa	CE	X=M(aaaa)					
	xx2	DE	X=M((X)+xx2)					
	xx1	EE	X=M((X)+xx1)					
	xx0	FE	X=M(X)					
LSL	aa	38	M(aa)=M(aa) A< 1	-	-	T	T	T
	xx1	68	M((X)+xx1)=M((X)+xx1) A< 1					
	xx0	78	M(X)=M(X) A< 1					
LSLA	---	48	A=(A) A< 1	-	-	T	T	T
LSLX	---	58	X=(X) A< 1	-	-	T	T	T
LSR	aa	34	M(aa)=M(aa) L> 1	-	-	0	T	T
	xx1	64	M((X)+xx1)=M((X)+xx1) L> 1					
	xx0	74	M(X)=M(X) L> 1					
LSRA	---	44	A=(A) L> 1	-	-	0	T	T
LSRX	---	54	X=(X) L< 1	-	-	0	T	T
NEG	aa	30	M(aa)=00-M(aa)	-	-	T	T	T
	xx1	60	M((X)+xx1)=00-M((X)+xx1)					
	xx0	70	M(X)=00-M(X)					
NEGA	---	40	A=00-(A)	-	-	T	T	T
NEGX	---	50	X=00-(X)	-	-	T	T	T
NOP	---	9D	P=(P)+0001	-	-	-	-	-
ORA	ii	AA	A=(A) or ii	-	-	T	T	-
	aa	BA	A=(A) or M(aa)					
	aaaa	CA	A=(A) or M(aaaa)					
	xx2	DA	A=(A) or M((X)+xx2)					
	xx1	EA	A=(A) or M((X)+xx1)					
	xx0	FA	A=(A) or M(X)					
ROL	aa	39	M(aa)=M(aa) R< 1	-	-	T	T	T
	xx1	69	M((X)+xx1)=M((X)+xx1) R< 1					
	xx0	79	M(X)=M(X) R< 1					
ROLA	---	49	A=(A) R< 1	-	-	T	T	T
ROLX	---	59	X=(X) R< 1			T	T	T
ROR	aa	36	M(aa)=M(aa) R> 1	-	-	T	T	T
	xx1	66	M((X)+xx1)=M((X)+xx1) R> 1					
	xx0	76	M(X)=M(X) R> 1					
RORA	---	46	A=(A) R> 1	-	-	T	T	T
RORX	---	56	X=(X) R> 1	-	-	T	T	T
RSP	---	9C	S=7F	-	-	-	-	-
RTI	---	80	Return from interrupt	?	?	?	?	?
RTS	---	81	Return from subroutine	-	-	-	-	-

Mne-monic	Oper-and	Op-code	Function	Status H I N Z C
SBC	ii	A2	A=(A)-ii-(C)	- - T T T
	aa	B2	A=(A)-M(aa)-(C)	
	aaaa	C2	A=(A)-M(aaaa)-(C)	
	xx2	D2	A=(A)-M((X)+xx2)-(C)	
	xx1	E2	A=)A)-M((X)+xx1)-(C)	
	xx0	F2	A=(A)-M(X)-(C)	
SEC	---	99	C=1	- - - - 1
SEI	---	9B	I=1	- 1 - - -
STA	aa	B7	M(aa)=(A)	- - T T -
	aaaa	C7	M(aaaa)=(A)	
	xx2	D7	M((X)+xx2)=(A)	
	xx1	E7	M((X)+xx1)=(A)	
	xx0	F7	M(X)=(A)	
STOP	---	8E	CMOS version only	- - - - -
STX	aa	BF	M(aa)=(X)	- - T T -
	aaaa	CF	M(aaaa)=(A)	
	xx2	DF	M((X)+xx2)=(X)	
	xx1	EF	M((X)+xx1)=(X)	
	xx0	FF	M(X)=(X)	
SUB	ii	A0	A=(A)-ii	- - T T T
	aa	B0	A=(A)-M(aa)	
	aaaa	C0	A=(A)-M(aaaa)	
	xx2	D0	A=(A)-M((X)+xx2)	
	xx1	E0	A=(A)-M((X)+xx1)	
	xx0	F0	A=(A)-M(X)	
SWI	---	83	Software interrupt	- T - - -
TAX	---	97	(X)=(A)	- - - - -
TST	aa	3D	M(aa)-00	- - T T -
	xx1	6D	M((X)+xx1)-00	
	xx0	7D	M(X)-00	
TSTA	---	4D	(A)-00	- - T T -
TSTX	---	5D	(X)-00	- - T T -
TXA	---	9F	(A)=(X)	- - - - -
WAIT	---	8F	CMOS version only	- - - - -

B.4 M6809 INSTRUCTIONS

In the following table, the "Function" column for branch and long branch instructions only contains the test condition performed by the branch. The following function will be performed if the result of the test is true:

P=(P)+0002+rr (for branch)

P=(P)+0003+rrrr (for 1-byte long branch opcode)

P=(P)+0004+rrrr (for 2-byte long branch opcode)

If the result of the test is false, the following function will be performed:

P=(P)+0002 (for branch)

P=(P)+0003 (for 1-byte long branch opcode)

P=(P)+0004 (for 2-byte long branch opcode)

The functions for the instructions BSR, CWAI, DAA, EXG, JSR, LBSR, PSHS, PSHU, PULS, PULU, RTI, RTS, SEX, SWI, SWI2, SWI3, SYNC, and TFR are described in detail in the M6809 Programming Reference Manual.

Mne-monic	Oper-and	Op-code	Function	F	H	I	N	Z	V	C
ABX	--	3A	X=(X)+(B)	-	-	-	-	-	-	-
ADCA	ii	89	A=(A)+ii+(C)	-	T	-	T	T	T	T
	aa	99	A=(A)+ M(aa)+(C)							
	xxop	A9	A=(A)+xxop+(C)							
	aaaa	B9	A=(A)+M(aaaa)+(C)							
ADCB	ii	C9	B=(B)+ii+(C)	-	T	-	T	T	T	T
	aa	D9	B=(B)+M(aa)+(C)							
	xxop	E9	B=(B)+xxop+(C)							
	aaaa	F9	B=(B)+M(aaaa)+(C)							
ADDA	ii	8B	A=(A)+ii	-	T	-	T	T	T	T
	aa	9B	A=(A)+M(aa)							
	xxop	AB	A=(A)+xxop							
	aaaa	BB	A=(A)+M(aaaa)							
ADDB	ii	CB	B=(B)+ii	-	T	-	T	T	T	T
	aa	DB	B=(B)+M(aa)							
	xxop	EB	B=(B)+xxop							
	aaaa	FB	B=(B)+M(aaaa)							
ADDD	iiii	C3	D=(D)+iiii	-	-	-	T	T	T	T
	aa	D3	D=(D)+M(aa,aa+1)							
	xxop	E3	D=(D)+xxop							
	aaaa	F3	D=(D)+M(aaaa,aaaa+1)							
ANDA	ii	84	A=(A) and ii	-	-	-	T	T	0	-
	aa	94	A=(A) and M(aa)							
	xxop	A4	A=(A) and xxop							
	aaaa	B4	A=(A) and M(aaaa)							
ANDB	ii	C4	B=(B) and ii	-	-	-	T	T	0	-
	aa	D4	B=(B) and M(aa)							
	xxop	E4	B=(B) and xxop							
	aaaa	F4	B=(B) and M(aaaa)							
ANDCC	ii	1C	CC=(CC) and ii	?	?	?	?	?	?	?
ASL	aa	08	M(aa)=M(aa) A< 1	-	?	-	T	T	?	T
	xxop	68	xxop=xxop A< 1							
	aaaa	78	M(aaaa)=M(aaaa) A< 1							
ASLA	--	48	A=(A) A< 1	-	?	-	T	T	?	T
ASLB	--	58	B=(B) A< 1	-	?	-	T	T	?	T
ASR	aa	07	M(aa)=M(aa) A> 1	-	?	-	T	T	?	T
	xxop	67	xxop=xxop A> 1							
	aaaa	77	M(aaaa)=M(aaaa) A> 1							
ASRA	--	47	A=(A) A> 1	-	?	-	T	T	?	T
ASRB	--	57	B=(B) A> 1	-	?	-	T	T	?	T
BCC	rr	24	Test (C)=0	-	-	-	-	-	-	-
BCS	rr	25	Test (C)=1	-	-	-	-	-	-	-
BEQ	rr	27	Test (Z)=1	-	-	-	-	-	-	-
BGE	rr	2C	Test (N) xor (V)=0	-	-	-	-	-	-	-
BGT	rr	2E	Test (Z) or [(N) xor (V)]=0	-	-	-	-	-	-	-
BHI	rr	22	Test (C) xor (Z)=0	-	-	-	-	-	-	-
BHS	rr	24	Test (C)=0	-	-	-	-	-	-	-

Mnemonic	Operand	Opcode	Function	Status F H I N Z V C
BITA	ii	85	(A) and ii	- - - T T 0 -
	aa	95	(A) and M(aa)	
	xxop	A5	(A) and xxop	
	aaaa	B5	(A) and M(aaaa)	
BITB	ii	C5	(B) and ii	- - - T T 0 -
	aa	D5	(B) and M(aa)	
	xxop	E5	(B) and xxop	
	aaaa	F5	(B) and M(aaaa)	
BLE	rr	2F	Test (Z) or [(N) xor (V)]=1	- - - - - - -
BLO	rr	25	TEST (C)=1	- - - - - - -
BLS	rr	23	Test (C) or (Z)=1	- - - - - - -
BLT	rr	2D	Test (N) xor (V)=1	- - - - - - -
BMI	rr	2B	Test (N)=1	- - - - - - -
BNE	rr	26	Test (Z)=0	- - - - - - -
BPL	rr	2A	Test (N)=0	- - - - - - -
BRA	rr	20	Tests always true	- - - - - - -
BRN	rr	21	Tests always false	- - - - - - -
BSR	rr	8D	Subroutine call	- - - - - - -
BVC	rr	28	Test (V)=0	- - - - - - -
BVS	rr	29	Test (V)=1	- - - - - - -
CLR	aa	0F	M(aa)=00	- - - 0 1 0 0
	xxop	6F	xxop=00	
	aaaa	7F	M(aaaa)=00	
CLRA	--	4F	A=00	- - - 0 1 0 0
CLRB	--	5F	B=00	- - - 0 1 0 0
CMPA	ii	81	(A)-ii	- ? - T T T T
	aa	91	(A)-M(aa)	
	xxop	A1	(A)-xxop	
	aaaa	B1	(A)-M(aaaa)	
CMPB	ii	C1	(B)-ii	- ? - T T T T
	aa	D1	(B)-M(aa)	
	xxop	E1	(B)-xxop	
	aaaa	F1	(B)-M(aaaa)	
CMPD	iiii	10,83	(D)-iiii	- - - T T T T
	aa	10,93	(D)-M(aa,aa+1)	
	xxop	10,A3	(D)-xxop	
	aaaa	10,B3	(D)-M(aaaa,aaaa+1)	
CMPS	iiii	11,8C	(S)-iiii	- - - T T T T
	aa	11,9C	(S)-M(aa,aa+1)	
	xxop	11,AC	(S)-xxop	
	aaaa	11,BC	(S)-M(aaaa,aaaa+1)	
CMPU	iiii	11,83	(U)-iiii	- - - T T T T
	aa	11,93	(U)-M(aa,aa+1)	
	xxop	11,A3	(U)-xxop	
	aaaa	11,B3	(U)-M(aaaa,aaaa+1)	
CMPX	iiii	8C	(X)-iiii	- - - T T T T
	aa	9C	(X)-M(aa,aa+1)	
	xxop	AC	(X)-xxop	
	aaaa	BC	(X)-M(aaaa,aaaa+1)	

Mnemonic	Operand	Opcode	Function	Status F H I N Z V C
CMPY	iiii	10,8C	(Y)-iiii	- - - T T T T
	aa	10,9C	(Y)-M(aa,aa+1)	
	xxop	10,AC	(Y)-xxop	
	aaaa	10,BC	(Y)-M(aaaa,aaaa+1)	
COM	aa	03	M(aa)=M(aa) xor $FF	- - - T T 0 1
	xxop	63	xxop=xxop xor $FF	
	aaaa	73	M(aaaa)=M(aaaa) xor $FF	
COMA	--	43	A=(A) xor $FF	- - - T T 0 1
COMB	--	53	B=(B) xor $FF	- - - T T 0 1
CWAI	ii	3C	Clear and wait for interrupt	? ? ? ? ? ? ?
DAA	--	19	Converts binary add of BCD into BCD	- - - T T T T
DEC	aa	0A	M(aa)=M(aa)-01	- - - T T ? -
	xxop	6A	xxop=xxop-01	
	aaaa	7A	M(aaaa)=M(aaaa)-01	
DECA	--	4A	A=(A)-01	- - - T T ? -
DECB	--	5A	B=(B)-01	- - - T T ? -
EORA	ii	88	A=(A) xor ii	- - - T T 0 -
	aa	98	A=(A) xor M(aa)	
	xxop	A8	A=(A) xor xxop	
	aaaa	B8	A=(A) xor M(aaaa)	
EORB	ii	C8	B=(B) xor ii	- - - T T 0 -
	aa	D8	B=(B) xor M(aa)	
	xxop	E8	B=(B) xor xxop	
	aaaa	F8	B=(B) xor M(aaaa)	
EXG	rl	1E	Exchange 2 registers	? ? ? ? ? ? ?
INC	aa	0C	M(aa)=M(aa)+01	- - - T T ? -
	xxop	6C	xxop=xxop+01	
	aaaa	7C	M(aaaa)=M(aaaa)+01	
INCA	--	4C	A=(A)+01	- - - T T ? -
INCB	--	5C	B=(B)+01	- - - T T ? -
JMP	aa	0E	P=aa	- - - - - - -
	xxop	6E	P=xxop	
	aaaa	7E	P=aaaa	
JSR	aa	9D	Subroutine call	- - - - - - -
	xxop	AD	Subroutine call	
	aaaa	BD	Subroutine call	
LBCC	rrrr	10,24	Test (C)=0	- - - - - - -
LBCS	rrrr	10,25	Test (C)=1	- - - - - - -
LBEQ	rrrr	10,27	Test (Z)=1	- - - - - - -
LBGE	rrrr	10,2C	Test (N) xor (V)=0	- - - - - - -
LBGT	rrrr	10,2E	Test (Z) or [(N) xor (V)]=0	- - - - - - -
LBHI	rrrr	10,22	Test (C) xor (Z)=0	- - - - - - -
LBHS	rrrr	10,24	Test (C)=0	- - - - - - -
LBLE	rrrr	10,2F	Test (Z) or [(N) xor (V)]=1	- - - - - - -
LBLO	rrrr	10,25	Test (C)=1	- - - - - - -
LBLS	rrrr	10,23	Test (C) or (Z)=1	- - - - - - -
LBLT	rrrr	10,2D	Test (N) xor (V)=1	- - - - - - -
LBMI	rrrr	10,2B	Test (N)=1	- - - - - - -

Mnemonic	Operand	Opcode	Function	F	H	I	N	Z	V	C
LBNE	rrrr	10,26	Test (Z)=0	-	-	-	-	-	-	-
LBPL	rrrr	10,2A	Test (N)=0	-	-	-	-	-	-	-
LBRA	rrrr	16	Tests always true	-	-	-	-	-	-	-
LBRN	rrrr	10,21	Tests always false	-	-	-	-	-	-	-
LBSR	rrrr	17	Subroutine call	-	-	-	-	-	-	-
LBVC	rrrr	10,28	Test (V)=0	-	-	-	-	-	-	-
LBVS	rrrr	10,29	Test (V)=1	-	-	-	-	-	-	-
LDA	ii	86	A=ii	-	-	-	T	T	0	-
	aa	96	A=M(aa)							
	xxop	A6	A=xxop							
	aaaa	B6	A=M(aaaa)							
LDB	ii	C6	B=ii	-	-	-	T	T	0	-
	aa	D6	B=M(aa)							
	xxop	E6	B=xxop							
	aaaa	F6	B=M(aaaa)							
LDD	iiii	CC	D=iiii	-	-	-	T	T	0	-
	aa	DC	D=M(aa,aa+1)							
	xxop	EC	D=xxop							
	aaaa	FC	D=M(aaaa,aaaa+1)							
LDS	iiii	10,CE	S=iiii	-	-	-	T	T	0	-
	aa	10,DE	S=M(aa,aa+1)							
	xxop	10,EE	S=xxop							
	aaaa	10,FE	S=M(aaaa,aaaa+1)							
LDU	iiii	CE	U=iiii	-	-	-	T	T	0	-
	aa	DE	U=M(aa,aa+1)							
	xxop	EE	U=xxop							
	aaaa	FE	U=M(aaaa,aaaa+1)							
LDX	iiii	8E	X=iiii	-	-	-	T	T	0	-
	aa	9E	X=M(aa,aa+1)							
	xxop	AE	X=xxop							
	aaaa	BE	X=M(aaaa,aaaa+1)							
LDY	iiii	10,8E	Y=iiii	-	-	-	T	T	0	-
	aa	10,9E	Y=M(aa,aa+1)							
	xxop	10,AE	Y=xxop							
	aaaa	10,BE	Y=M(aaaa,aaaa+1)							
LEAS	xxop	32	S=effad xxop	-	-	-	-	-	-	-
LEAU	xxop	33	U=effad xxop	-	-	-	-	-	-	-
LEAX	xxop	30	X=effad xxop	-	-	-	-	T	-	-
LEAY	xxop	31	Y=effad xxop	-	-	-	-	T	-	-
LSL	aa	08	M(aa)=M(aa) A< 1	-	?	-	T	T	?	T
	xxop	68	xxop=xxop A< 1							
	aaaa	78	M(aaaa)=M(aaaa) A< 1							
LSLA	--	48	A=(A) A< 1	-	?	-	T	T	?	T
LSLB	--	58	B=(B) A< 1	-	?	-	T	T	?	T
LSR	aa	04	M(aa)=M(aa) L> 1	-	-	-	0	T	-	T
	xxop	64	xxop=xxop L> 1							
	aaaa	74	M(aaaa)=M(aaaa) L> 1							

Mnemonic	Operand	Opcode	Function	Status F H I N Z V C
LSRA	--	44	A=(A) L> 1	- - - 0 T - T
LSRB	--	54	B=(B) L> 1	- - - 0 T - T
MUL	--	3D	D=(A)*(B)	- - - - T - T
NEG	aa	00	M(aa)=00-M(aa)	- ? - T T ? T
	xxop	60	xxop=00-xxop	
	aaaa	70	M(aaaa)=00-M(aaaa)	
NEGA	--	40	A=00-(A)	- ? - T T ? T
NEGB	--	50	B=00-(B)	- ? - T T ? T
NOP	--	12	P=(P)+0001	- - - - - - -
ORA	ii	8A	A=(A) or ii	- - - T T 0 -
	aa	9A	A=(A) or M(aa)	
	xxop	AA	A=(A) or xxop	
	aaaa	BA	A=(A) or M(aaaa)	
ORB	ii	CA	B=(B) or ii	- - - T T 0 -
	aa	DA	B=(B) or M(aa)	
	xxop	EA	B=(B) or xxop	
	aaaa	FA	B=(B) or M(aaaa)	
ORCC	ii	1A	CC=(CC) or ii	? ? ? ? ? ? ?
PSHS	rl	34	Push registers on M(S)	- - - - - - -
PSHU	rl	36	Push registers on M(U)	- - - - - - -
PULS	rl	35	Pull registers from M(S)	? ? ? ? ? ? ?
PULU	rl	37	Pull registers from M(U)	? ? ? ? ? ? ?
ROL	aa	09	M(aa)=M(aa) R< 1	- - - T T ? T
	xxop	69	xxop=xxop R< 1	
	aaaa	79	M(aaaa)=M(aaaa) R< 1	
ROLA	--	49	A=(A) R< 1	- - - T T ? T
ROLB	--	59	B=(B) R< 1	- - - T T ? T
ROR	aa	06	M(aa)=M(aa) R> 1	- - - T T ? T
	xxop	66	xxop=xxop R> 1	
	aaaa	76	M(aaaa)=M(aaaa) R> 1	
RORA	--	46	A=(A) R> 1	- - - T T ? T
RORB	--	56	B=(B) R> 1	- - - T T ? T
RTI	--	3B	Return from interrupt	? ? ? ? ? ? ?
RTS	--	39	Return from subroutine	- - - - - - -
SBCA	ii	82	A=(A)-ii-(C)	- - - T T T T
	aa	92	A=(A)-M(aa)-(C)	
	xxop	A2	A=(A)-xxop-(C)	
	aaaa	B2	A=(A)-M(aaaa)-(C)	
SBCB	ii	C2	B=(B)-ii-(C)	- - - T T T T
	aa	D2	B=(B)-M(aa)-(C)	
	xxop	E2	B=(B)-xxop-(C)	
	aaaa	F2	B=(B)-M(aaaa)-(C)	
SEX	--	1D	Sign extension of B into A	- - - T T 0 -
STA	aa	97	M(aa)=(A)	- - - T T 0 -
	xxop	A7	xxop=(A)	
	aaaa	B7	M(aaaa)=(A)	

Mne-monic	Oper-and	Op-code	Function	Status F H I N Z V C
STB	aa	D7	M(aa)=(B)	- - - T T 0 -
	xxop	E7	xxop=(B)	
	aaaa	F7	M(aaaa)=(B)	
STD	aa	DD	M(aa,aa+1)=(D)	- - - T T 0 -
	xxop	ED	xxop=(D)	
	aaaa	FD	M(aaaa,aaaa+1)=(D)	
STS	aa	10,DF	M(aa,aa+1)=(S)	- - - T T 0 -
	xxop	10,EF	xxop=(S)	
	aaaa	10,FF	M(aaaa,aaaa+1)=(S)	
STU	aa	DF	M(aa,aa+1)=(U)	- - - T T 0 -
	xxop	EF	xxop=(U)	
	aaaa	FF	M(aaaa,aaaa+1)=(U)	
STX	aa	9F	M(aa,aa+1)=(X)	- - - T T 0 -
	xxop	AF	xxop=(X)	
	aaaa	BF	M(aaaa,aaaa+1)=(X)	
STY	aa	10,9F	M(aa,aa+1)=(Y)	- - - T T 0 -
	xxop	10,AF	xxop=(Y)	
	aaaa	10,BF	M(aaaa,aaaa+1)=(Y)	
SUBA	ii	80	A=(A)-ii	- ? - T T T T
	aa	90	A=(A)-M(aa)	
	xxop	A0	A=(A)-xxop	
	aaaa	B0	A=(A)-M(aaaa)	
SUBB	ii	C0	B=(B)-ii	- ? - T T T T
	aa	D0	B=(B)-M(aa)	
	xxop	E0	B=(B)-xxop	
	aaaa	F0	B=(B)-M(aaaa)	
SUBD	iiii	83	D=(D)-iiii	- - - T T T T
	aa	93	D=(D)-M(aa,aa+1)	
	xxop	A3	D=(D)-xxop	
	aaaa	B3	D=(D)-M(aaaa,aaaa+1)	
SWI	--	3F	Software interrupt	- - - - - - -
SWI2	--	10,3F	Software interrupt	- - - - - - -
SWI3	--	11,3F	Software interrupt	- - - - - - -
SYNC	--	13	Synchronize	- - - - - - -
TFR	rl	1F	Transfer register	? ? ? ? ? ? ?
TST	aa	0D	M(aa)-00	- - - T T 0 -
	xxop	6D	xxop-00	
	aaaa	7D	M(aaaa)-00	
TSTA	--	4D	(A)-00	- - - T T 0 0
TSTB	--	5D	(B)-00	- - - T T 0 0

B.5 M6809 INDEXED ADDRESSING MODES

The value of the post-byte (the first byte following the opcode) for instructions using the indexed addressing mode is determined by the format of the operand. Two formats exist: simple indexing and complex indexing. Simple indexing is used when the operand is of the form:

<exp>,R

where <exp> is an absolute expression in the range -16 to 15 but not equal to zero, and R is one of the index registers "S", "U", "X", or "Y". All other indexed addressing modes use the complex indexing format. The two post-byte formats are described below:

Simple Indexing -- Post-Byte

7	6	5	4	3	2	1	0
0	RR		OFFSET				

```
where RR=00 if X register
         01 if Y register
         10 if U register
         11 if S register

      OFFSET=5-bit 2's complement
```

Complex Indexing -- Post-Byte

7	6	5	4	3	2	1	0
1	RR		I	TTTT			

```
where  RR= 00 if X or PCR
           01 if Y
           10 if U
           11 if S

       I=  0 if no indirect
           1 if indirect

    TTTT=0000   Single auto-increment (R+)
         0001   Double auto-increment (R++)
         0010   Single auto-decrement (-R)
         0011   Double auto-decrement (--R)
         0100   0 offset value or no offset
         0101   Accumulator B is offset (B,R)
         0110   Accumulator A is offset (A,R)
         1000   8-bit offset
         1001   16-bit offset
         1011   Accumulator D is offset (D,R)
         1100   8-bit offset with PCR
         1101   16-bit offset with PCR
         1111   Extended indirect
```

B.6 M6800/M6801 INSTRUCTIONS AND M6809 EQUIVALENTS

Not all M6800/M6801 instructions have exact equivalences recognized by the M6809 Macro Assembler. Some translate into instructions that generate more bytes by the M6809 Macro Assembler. However, all opcode mnemonics recognized by the M6800/M6801 Macro Assembler are recognized by the M6809 Macro Assembler, and are translated into equivalent M6809 code where possible. Some translations are not equivalent, but the same function is still performed. In addition, some "M6800-like" mnemonics are recognized by the M6809 Macro Assembler and translated.

M6800/M6801 Mnemonic	Type of Instruction	M6809 Equivalent
ABA	6800	PSHS B ADDA S+
ASLD	6801	ASLB ROLA
CBA	6800	PSHS B CMPA S+
CLC	6800	ANDCC #$FE
CLF	6800-like	ANDCC #$BF
CLI	6800	ANDCC #$EF
CLIF	6800-like	ANDCC #$AF
CLV	6800	ANDCC #$FD
CPX	6800	CMPX
DES	6800	LEAS -1,S
DEX	6800	LEAX -1,X
DEY	6800-like	LEAY -1,Y
INS	6800	LEAS 1,S
INX	6800	LEAX 1,X
INY	6800-like	LEAY 1,Y
LDAA; LDA A	6800	LDA
LDAB; LDA B	6800	LDA
LDAD	6801	LDD
LSLD	6801	ASLB ROLA
LSRD	6801	LSRA RORB
ORAA; ORA A	6800	ORA
ORAB; ORA B	6800	ORB
PSHA; PSH A	6800	PSHS A
PSHB; PSH B	6800	PSHS B
PSHX	6801	PSHS X
PULA; PUL A	6800	PULS A
PULB; PUL B	6800	PULS B
PULX	6801	PULS X
SBA	6800	PSHS B SUBA S+

M6800/M6801 Mnemonic	Type of Instruction	M6809 Equivalent
SEC	6800	ORCC #$01
SEF	6800-like	ORCC #$40
SEI	6800	ORCC #$10
SEIF	6800-like	ORCC #$50
SEV	6800	ORCC #$02
STAA; STA A	6800	STA
STAB; STA B	6800	STB
STAD	6801	STD
TAB	6800	TFR A,B TSTA
TBA	6800	TFR B,A TSTA
TAP	6800	TFR A,CC
TPA	6800	TFR CC,A
TSX	6800	TFR S,X
TXS	6800	TFR X,S
WAI	6800	CWAI #$FF

APPENDIX C

DIRECTIVE SUMMARY

A complete description of all directives appears in Chapter 4.

ASSEMBLY CONTROL

END	Program end
FAIL	Programmer generated errors
NAM	Assign program name
ORG	Origin program counter
SETDP	Set direct page pseudo register (M6809 only)

SYMBOL DEFINITION

ENDM	Macro definition end
EQU	Assign permanent value
MACR	Macro definition start
REG	Register list definition (M6809 only)
SET	Assign temporary 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
RMB	Reserve memory; single bytes

PROGRAM RELOCATION

ASCT	Absolute section
BSCT	Base section
COMM	Named common section
CSCT	Blank common section
DSCT	Data section
IDNT	Identification record
PSCT	Program section
OPT REL	Relocatable output selected
XDEF	External symbol definition
XREF	External symbol reference

CONDITIONAL ASSEMBLY

ENDC	End of current level of conditional assembly
IFC	Assemble if strings compare
IFEQ	Assemble if expression is equal to zero
IFGE	Assemble if expression is greater than or equal to zero
IFGT	Assemble if expression is greater than zero
IFLE	Assemble if expression is less than or equal to zero
IFLT	Assemble if expression is less than zero
IFNC	Assemble if strings do not compare
IFNE	Assemble if expression is not equal to zero

LISTING CONTROL

OPT ABS	Select absolute MDOS-loadable object output
OPT CL	Print conditional assembly directives
OPT NOCL	Don't print conditional assembly directives
OPT CMO	Allow CMOS instructions STOP and WAIT (M6805 only)
OPT NOCMO	Don't allow CMOS instructions STOP and WAIT (M6805 only)
OPT CRE	Print cross reference talbe
OPT G	Print generated lines of FCB, FCC, and FDB directives
OPT NOG	Don't print generated lines of FDB, FCC, and FDB directives
OPT L	Print source listing from this point
OPT NOL	Inhibit printing of source listing from this point
OPT LLE=n	Change line length
OPT LOAD	Select absolute EXORciser-loadable object output
OPT M	Create object output in memory
OPT MC	Print macro calls
OPT NOMC	Don't print macro calls
OPT MD	Print macro definitions
OPT NOMD	Don't print macro definitions
OPT MEX	Print macro expansions
OPT NOMEX	Don't print macro expansions
OPT O	Create object output file
OPT NOO	Do not create object output file
OPT P=n	Change page length
OPT NOP	Inhibit paging and printing of headings
OPT REL	Select relocatable object output

OPT S	Print symbol table
OPT SE	Print user-supplied sequence numbers
OPT U	Print unassembled code from conditional directives
OPT NOU	Don't print unassembled code from conditional directives
OPT W	Print warnings (M6809 only)
OPT NOW	Don't print warnings (M6809 only)
OPT ZØ1	Allow M6801 instruction mnemonics (M6800 only)
OPT NOZØ1	Don't allow M6801 instruction mnemonics (M6800 only)
PAGE	Print subsequent statements on top of next page
SPC	Skip lines
TTL	Initialize heading for source listing

APPENDIX D

ASSEMBLER MESSAGES

A description of all error and warning messages follows. Warning messages are only supported by the M6809 Macro Assembler. Some error messages only occur when using the M6809 Macro Assembler or the M6805 Macro Assembler. The format of the error is:

****ERROR XXX-- YYYYY

where XXX is the error message number, and YYYYY is the line number of the previously encountered error. If YYYYY = 00000, this indicates that there is no previous error. The format of the warning messages is similar. The EDOS and tape versions of the M6800 Macro Assembler do not include the line number of the last error.

D.1 ERROR MESSAGES

169 Invalid bit number (M6805 only)
The bit number in bit set/clear and bit test and branch instructions must be an absolute number in the range 0-7.

173 Invalid use of direct mode indicator (M6809 only)
The direct mode indicator, "<", was specified in the extended indirect addressing mode (e.g., LDA <[VAR]). The "<" is ignored.

174 Invalid auto increment/decrement format (M6809 only)
Single auto increment or decrement was specified in the indirect mode (e.g., LDA [X+]) or more than two minus or plus signs detected (e.g., LDA ---X).

175 Invalid index register format (M6809 only)
One of the accumulators "A", "B", or "D" was specified as the offset in the indexed mode, but was not followed by one of the index registers "S", "U", "X", or "Y" (e.g., LDA A,PCR).

176 Invalid expression for PSH/PUL (M6809 only)
The immediate expression following one of the instructions PSHS, PULS, PSHU, or PULU contained symbols defined with other than the REG directive (Paragraph 4.27), contained an operator other than "!+", or contained no symbols following the "#" (e.g., PSHU #$FF; PSHS #REG1*REG2).

177 Incompatible register for PSH/PUL instruction (M6809 only)
The register list for the PSHS/PULS instructions cannot contain the register "S", and the register list for the PSHU/PULU instructions cannot contain the register "U". The register list specified with the REG directive cannot contain both "U" and "S". In the case with the REG directive, the value assigned to the symbol will be the first "U" or "S" encountered (e.g., PSHS S).

178 Invalid register operand specification (M6809 only)
Undefined register name encountered in register list; not exactly two register names in register list specification for TFR or EXG instructions; or no register list specified for PSH/PUL instructions. VAlid register names are: A, B, CC, D, DP, PC, S, U, X, and Y (e.g., TFR A,B,X; PULU Q).

179 Incompatible register pair (M6809 only)
The register pair of an EXG instruction was not the same size (i.e., two 16-bit registers or two 8-bit registers), or the register pair specification of a TFR instruction indicated a transfer from an 8-bit register to a 16-bit register. The 8-bit registers are: "A", "B", "CC", and "DP". The 16-bit registers are: "D", "PC", "S", "U", "X", and "Y" (e.g., EXG X,A; TFR B,PC).

202 Label or opcode error
The label or opcode symbol does not begin with an alphabetic character or a period.

205 Label error
The statement label field is not terminated with a blank. This usually occurs if an invalid character is used in the label.

207 Undefined opcode
The symbol in the opcode field is not a valid opcode mnemonic, directive, or macro definition.

208 Branch out of range
The operand resulted in an offset greater than 129 bytes forward or 126 bytes backward from the first byte of the branch instruction. This error may also occur if the operand is in a different program section (relocatable) than the current program counter section.

209 Illegal addressing mode
The specified addressing mode in the operand field is not valid with this instruction type.

210 Byte overflow -- operand too large
The operand's value exceeded 1 byte (8 bits). The most significant eight bits of the 16-bit expression must be all zeros or all ones for a one-byte field.

211 Undefined symbol
The symbol never appears in a label field.

212 Directive operand error
A syntax error was detected in the operand field of a directive.

214 FCB directive syntax error
The structure of the FCB directive is syntactically incorrect.

215 FDB directive operand error
The structure of the FDB directive is syntactically incorrect.

216 Directive operand error
The directive's operand field is missing, terminated by an invalid terminator, or an expression in the operand field contains an invalid operator.

217 Option error
An option in the operand field of the OPT directive was undefined.

219 No END statement

The END directive was not found at the end of the last source file. The END directive is automatically supplied.

220 Phasing error

The value of the program counter during pass 1 and pass 2 for the same instruction is different.

221 Symbol table or macro table overflow

The symbol table or macro table has overflowed. This is a fatal error, and terminates the Assembler during pass 1.

222 Reserved symbol used

One of the reserved symbols (A, B, or X) appeared in the label field or in the operand field of a statement. These symbols can only be used in the operation field to modify the root mnemonic (A or B) or in the operand field to specify indexed addressing (e.g., ,X). For the M6809 Macro Assembler, other reserved symbols are Y, U, S, D, CC, DP, PC, and PCR. For the M6805 Macro Assembler, only A and X are reserved symbols.

223 The directive must or must not have a label

Depending on the directive used, the label field must be blank or must contain a valid symbol.

225 Named common name used in expression

A named common section name can only appear in the label field of another COMM directive. Its use anywhere else is invalid.

226 Illegal parenthesis

The parentheses in an expression do not balance.

227 Too many digits in numeric constant

An overflow in the numeric evaluation of a constant was detected. Also used if a sequence number is missing on a line in a file that has sequence numbers.

228 Invalid usage of operator

The multiplication, division, and two-character operators cannot be used in a relocatable expression or with external references.

229 Invalid starting execution address

The starting execution address specified as the expression on the END statement is not within the range of the MDOS-loadable object file. This can happen, since RMB's at the beginning or end of the program are not included in the range of the program.

230 CSCT initialization error

No initialized code can be placed into CSCT.

231 Multiple relocatable section types

More than one relocatable section type occurred in the evaluation of an expression or one relocatable symbol occurred with a unary minus preceding it.

232 Relocation count error
The relocation count for a given section after an expression evaluation was greater than one (e.g., adding two PSCT symbols).

233 Symbol name too large
A symbol of greater than 6 characters was encountered.

234 Multiply defined symbol
A reference was made to a multiply defined symbol.

235 Memory error
The OPT M option was used and object code was going to be written into non-existent memory or into contiguous memory belonging to the Assembler.

236 Program counter overflow
The program counter overflowed its maximum value for a particular section ($FF for BSCT, $FFFF for all other sections).

237 Invalid terminator for sequence number
The character following a user-supplied sequence number was not a blank.

238 Section table overflow
Too many ASCT and named common sections were specified. This is a fatal error, and terminates the Assembler during pass 1.

239 Illegal directive in absolute mode
A relocation directive (e.g., PSCT, COMM, etc.) was used, but the relocation option (OPT REL) was not specified.

240 Inconsistent or invalid named common operand
The operand field of the COMM directive did not contain BSCT, DSCT, or PSCT specifications; or the operand field was different from the one used the first time.

241 Illegal symbol used in an expression
An undefined forward reference, external reference, or relocatable symbol was used illegally in an expression. The instruction will not be relocated by the M6800 Linking Loader.

242 OPT directive error
The "LOAD", "REL", or "ABS" options were used in combination; the "REL" option was not ASCT or the program counter was not zero; or the "CRE" option was used after the first symbol had already been placed into the symbol table.

243 XREF or XDEF directive operand error
An invalid symbol or no operand was detected in the operand field of the XDEF or XREF directive.

244 Illegal page or listing line length
A page or listing line length was not within the allowed range.

245 Invalid use of common variable
A variable in blank or named common cannot be used in the operand field of the XDEF, XREF, or COMM directive.

247 Invalid terminator for an operand
The character following the legal part of an operand is not a valid terminator (usually a carriage return or space). For the M6809 Macro Assembler, this error could occur if invalid indirect pairing; i.e., an operand has "[" but no "]".

248 Macro definition error
An attempt was made to define a macro that already existed.

249 Macro parenthesis error
Parentheses in macro call argument are not balanced.

250 Macro definition nest error
A macro directive was encountered during a macro expansion. Macro definitions cannot be nested.

251 Macro expansion nest error
Macro calls were nested too deep, or the number of ENDM directives does not match the number of MACR directives.

252 Invalid macro argument index
The character following a backslash (\) during macro expansion was not an alphanumeric or a period.

253 IFC, IFNC directive syntax error
No operand was found or no comma was found to separate the two arguments.

254 Conditional directives nest error
Conditional directives were nested too deep, or the number of ENDC directives did not match the number of IFxx directives.

255 FAIL directive warning
The FAIL directive (a planned program error) was encountered.

D.2 M6809 WARNING MESSAGES

1 Long branch not required

A long branch instruction was used to branch to an address within the range -126 to +129. Although the long branch instruction could be changed to a short branch, it could result in other out-of-range short branches.

2 Extended addressing should be used

Direct addressing was forced by using the "<" indicator. However, the direct page pseudo register assigned by the SETDP directive (Paragraph 4.27) indicated that the extended mode should have been used.

3 Duplicate register specification

The same register name was specified more than once in a register list. Register "D" specified with either register "A" or "B" gives this warning.

4 Possible SETDP expression error

The most significant byte of the expression in a SETDP directive was not zero. The direct page pseudo register is assigned the value of the least significant byte anyway.

5 Extended addressing should be used

Direct addressing was forced by using the "<" indicator with a CSCT, DSCT, or PSCT non-external expression. The expression will not be relocated by the M6800 Linking Loader.

6 Possible transfer error

The TFR instruction was used with a transfer from a 16-bit register to an 8-bit register. The result of such a transfer is to move the least significant byte of the 16-bit register to the 8-bit register.

APPENDIX E

ASSEMBLER OUTPUT FORMAT

All the numeric information printed on the source listing is in hexadecimal, unless otherwise noted.

E.1 M6800/M6801 FORMAT

The MDOS version of the Macro Assembler will automatically print user-supplied sequence numbers in the left margin if they appear in the source file. However, the EDOS and tape versions of the Assembler will only print sequence numbers under control of the OPT directive. Then the sequence numbers will be printed in the right-most five columns of the source listing. Thus, the column titled "SEQ #" in the following table does not apply to EDOS and tape versions of the Macro Assembler.

COLUMN SEQ #	COLUMN NO SEQ #	CONTENTS
1-5	---	USER-SUPPLIED SEQUENCE NUMBER (DECIMAL)
7-11	1-5	SOURCE LINE NUMBER; A FIVE-DIGIT DECIMAL COUNTER MAINTAINED BY THE ASSEMBLER
12	6	PROGRAM COUNTER SECTION FLAG (A=ASCT, B=BSCT, C=CSCT, D=DSCT, N=NAMED COMMON, P=PSCT)
14-17	8-11	CURRENT PROGRAM COUNTER
19-20	13-14	MACHINE OPERATION CODE
		FOR NON-BRANCH INSTRUCTIONS:
22-23	16-17	FIRST BYTE OF OPERAND
24-25	18-19	SECOND BYTE OF OPERAND (IF ANY)
28	22	OPERAND SECTION FLAG (A, B, C, D, N, P)
		FOR BRANCH INSTRUCTIONS:
22-23	16-17	RELATIVE BRANCH OFFSET
25-28	19-22	ABSOLUTE ADDRESS OF DESTINATION
		FOR DIRECTIVES LIKE BSZ, EQU, ORG, ETC:
25-28	19-22	VALUE OF EXPRESSION
30-35	24-29	LABEL FIELD
37-42	31-36	OPERATION FIELD
43-50	37-44	OPERAND FIELD; LONGER OPERANDS EXTEND INTO THE COMMENT FIELD
52-132	46-132	COMMENT FIELD

E.2 M6805 FORMAT

The M6805 Macro Assembler will automatically print user-supplied sequence numbers in the left margin if they appear in the source file.

COLUMN Seq #	No Seq #	CONTENTS
1-5	---	User-supplied sequence number (decimal)
7-11	1-5	Source line number: a five-digit decimal counter maintained by the assembler
12	6	Program counter section flag (A=ASCT, B=BSCT, C=CSCT, D=DSCT, N=Named Common, P=PCST)
14-17	8-11	Current program counter
19-20	13-14	Machine operation code
		For non-branch instructions:
22-23	16-17	First byte of operand
24-25	18-19	Second byte of operand (if any)
31	25	Operand section flag (A,B,C,D,N,P)
		For branch instruction:
22-23	16-17	Relative branch offset
28-31	22-25	Absolute address of destination
		For bit test and branch instructions:
22-23	16-17	First byte of operand
25-26	19-20	Relative branch offset
28-31	22-25	Absolute address of destination
		For directives like BSZ, EQU, ORG, etc:
28-31	22-25	Value of expression
33-38	27-32	Label field
40-45	34-39	Operation field
46-53	40-47	Operand field: longer operands extend into the comment field
55-132	49-132	Comment field

E.3 M6809 FORMAT

The M6809 Macro Assembler will automatically print user-supplied sequence numbers in the left margin if they appear in the source file.

COLUMN Seq #	No Seq #	CONTENTS
1-5	---	User-supplied sequence number (decimal)
7-11	1-5	Source line number; a five-digit decimal counter maintained by the assembler
12	6	Program counter section flag (A=ASCT, B=BSCT, C=CSCT, D=DSCT, N=Named Common, P=PSCT)
14-17	8-11	Current program counter
19-20	13-14	First byte of machine operation code
21-22	15-16	Second byte of op-code (if any)
		For non-branch, non-indexed instructions:
24-25	18-19	First byte of operand
26-27	20-21	Second byte of operand (if any)
32	26	Operand section flag (A, B, C, D, N, P)
		For non-branch, indexed instructions:
24-25	18-19	Index post-byte
27-28	21-22	First byte of operand
29-30	23-24	Second byte of operand (if any)
32	26	Operand section flag
		For branch instructions:
24-25	18-19	First byte of relative branch offset
26-27	20-21	Second byte of offset (if any)
29-32	23-26	Absolute address of destination
		For M6800 equivalent instructions:
24-25	18-19	Second byte of translated instruction
27-28	21-22	Third byte of instruction (if any)

29-30	23-24	Fourth byte of instruction (if any)
		For directives like BSZ, EQU, ORG, etc:
29-32	23-26	Value of expression
34-39	28-33	Label field
41-46	35-40	Operation field
47-54	41-48	Operand field: longer operands extend into the comment field
56-132	50-132	Comment field

E.4 CROSS REFERENCE FORMAT

COLUMN	CONTENTS
1	Symbol Type Flag: D - External definition N - Named common symbol R - External reference U - Undefined symbol M - Multiply defined symbol S - "SET" symbol blank - None of the above
2	Symbol Section Flag blank - ASCT B - BSCT C - CSCT D - DSCT P - PSCT
4-7	Hexadecimal value of symbol
9-14	Symbol name
16-?	Assembler-maintained source line numbers of symbol reference. The asterisk appears after the line number of a symbol's definition. If the symbol was undefined, the asterisk will appear after the symbol's last reference.

APPENDIX F

M6800 MACRO ASSEMBLER/M6800 ASSEMBLER DIFFERENCES

Several differences exist between the M6800 Macro Assembler and the M6800 Co-resident Assembler. Obvious differences include such things as relocation, external references, external definitions, conditional assembly, extended expression evaluation (operators and parentheses), printing of titles on the source listing, printing of sequence numbers on the left side of the listing, macro definitions, and the M6801 instruction mnemonics.

Other differences are not attributable to major new features of the Macro Assembler. These differences include:

1. The "OPT O" option is no longer required to generate an object file. The object file is created as a default.

2. All expressions follow the normal rules of algebra rather than the strict left-to-right evaluation performed by the Co-resident Assembler.

3. The NAM directive is not required.

4. The symbol table space required for each symbol has changed from eight to ten bytes. In addition, if the cross reference option is in effect, an additional ten bytes are required for every four references to a symbol.

5. The Macro Assembler requires more memory.

6. In certain versions of the Macro Assembler, all of the Assembler options specified with the OPT directive can be specified on the command line that invokes the Assembler. This feature allows various options to be included or excluded without having to edit the source file.

7. Some versions also allow the source listing to be directed to a diskette file and to direct the printing of error messages to the printer (no listing being produced).

With the exception of the M6801 option, all of the above differences also apply to the M6805 and M6809 Macro Assemblers.

APPENDIX G

USING THE MACRO ASSEMBLER

The following paragraphs describe how to invoke the Macro Assembler from an MDOS diskette, an EDOS diskette, or from tape. Each section also includes an example of the command line format. After the Macro Assembler has been invoked, it will display a message of the following format:

```
MDOS MACROASSEMBLER 03.00
COPYRIGHT BY MOTOROLA 1977

M6800 MACROASSEMBLER 2.2
COPYRIGHT BY MOTOROLA 1978

M6805 MACROASSEMBLER 03.00
COPYRIGHT BY MOTOROLA 1978

M6809 MACROASSEMBLER 03.01
COPYRIGHT BY MOTOROLA 1978
```

to indicate the version of the assembler (M6800 MDOS - first sign on display; M6800 EDOS or tape - second sign on display; M6805 MDOS - third sign on display; M6809 MDOS - fourth sign on display) and the current revision number of the assembler.

G.1 M6800/M6801 MDOS MACRO ASSEMBLER

The M6800 Macro Assembler is invoked from the MDOS command line, as are other MDOS commands. However, the M6800 Macro Assembler requires that the system has a minimum of 24K bytes of memory. The format of the command line is:

```
RASM <name 1>[,<name 2>,...,<name n>] [;<options>]
```

where <name i> are the names of source files. Each file name in the list of source files is in the standard MDOS file name format:

```
<filename> [.<suffix>] [:<logical unit number>]
```

The default values of "SA" and "Ø" are used if suffix and logical unit number are not explicitly entered. Up to twenty file names can be accommodated by the Assembler. If multiple source files are specified, only the last source file should contain the END directive. If an END directive is found in a file prior to the last one, the assembly will exclude any files after the END directive.

The <options> may be one or more of the options listed in the following table. All options except those that control the destination of the source listing, the destination of the object file, and the printing of error messages on the printer if no listing is desired, can be specified from within the source program with the OPT directive. Certain options are automatically used as a default condition. These conditions can be reversed or overridden by preceding the option letter with a minus sign (-). The following options are recognized by the Assembler:

OPTION	DEFAULT	ATTRIBUTE CONTROLLED BY OPTION
A	-A	Absolute MDOS-loadable object file output
C	C	Printing of macro calls
D	D	Printing of macro definitions
E	-E	Printing of macro expansions
F	F	Printing of conditional directives
G	-G	Printing of generated code from FCB, FDB, and FCC directives
H	-H	Input initial heading from the console
L	-L	Print source listing on line printer
L=#CN,	-L	Print source listing on console
L=<name>,	-L	Print source listing into diskette file <name> (default suffix is "AL"; default logical unit is zero)
M	-M	Print error messages only on line printer
N=ddd,	N=72	Set printed line length to "ddd" (decimal)
O	O	Create object file with name <name 1> and suffix "LX" (absolute EXORciser-loadable), suffix "RO" (relocatable), or "LO" (absolute MDOS-loadable) on same drive as <name 1> of command line
O=<name>,	O	Create object file with name <name>
P=dd,	P=58	Set number of printed lines per page to "dd" (decimal)
R	-R	Relocatable object file output
S	-S	Print symbol table
U	-U	Print unassembled code between conditional directives
X	-X	Print cross reference table
Z	-Z	Use M6801 instruction mnemonics instead of M6800 and create M6801 object output

Certain options (L=, N=, O=, P=) require a terminating comma only if other options follow. Options are normally specified without any intervening blanks or separators. The options "L" and "M" are mutually exclusive, as are "A" and "R". The "A" option is only supported by the MDOS version of the Macro Assembler.

Each symbol in the symbol table requires ten bytes. Thus, if the minimum of 24K bytes of memory is used, the Macro Assembler can accommodate about 195 (decimal) symbols. However, if the cross reference option is specified, the symbol table requirements differ. In this case, an additional ten bytes are required by each symbol for every four references to that symbol. If macro definitions are used (MACR directive), the available symbol table space will be smaller.

Like other MDOS commands, the RASM command is sensitive to the BREAK and CTL-W keys of the system console.

The following are examples of valid MDOS command lines that invoke the Macro Assembler:

RASM SFILE1;LRX

This command line causes the Macro Assembler to assemble the source file SFILE1.SA:Ø in the relocatable mode ("R" option). A source listing will be directed to the system line printer ("L" option). At the end of the source listing, a cross reference table will be printed ("X" option). An object output file, SFILE1.RO:Ø, will also be produced automatically.

RASM FILEA:1;O=TEMP:Ø

This command line causes the Macro Assembler to assemble the source file FILEA.SA:1. No source listing will be generated, regardless of the OPT L directives within the source file. An object file will be created on drive zero. The suffix of the file will be "LX" (if no OPT REL or OPT ABS is contained in source file) or "RO" (if OPT REL is contained in source file) or "LO" (if OPT ABS is contained in source file.

RASM F1,F2,F3:1;L-OS

This command line causes the Macro Assembler to assemble the three source files F1.SA:Ø, F2.SA:Ø, and F3.SA:1 as if they were one contiguous source file. A source listing is produced on the system line printer. No object output file will be created. A symbol table will be printed at the end of the source listing.

RASM TEST;A

This command line causes the Macro Assembler to assemble the source file TEST.SA:Ø. No source listing will be generated. An object file will be created on drive zero (Ø). Its name will be TEST.LO, and it will be in a format that can be loaded by MDOS.

G.2 M6805 MACRO ASSEMBLER

The M6805 Macro Assembler only runs under MDOS. It is invoked from the MDOS command line, as are other MDOS commands. The format of the command line is:

```
RASM05 <name 1>[,<name 2>,...,<name n>] [;<options>]
```

With the following exceptions, the command line parameters are the same as described for the M6800 MDOS Macro Assembler (Paragraph G.1).

1. The "Z" option does not exist.
2. With 24K bytes of memory, the M6805 Macro Assembler can accommodate about 185 (decimal) symbols.

G.3 M6809 MACRO ASSEMBLER

The M6809 Macro Assembler only runs under MDOS. It is invoked from the MDOS command line, as are other MDOS commands. However, the M6809 Macro Assembler requires that the system has a minimum of 32K bytes of memory. The format of the command line is:

```
RASM09 <name 1>[,<name 2>,...,<name n>] [;<options>]
```

With the following exceptions, the command line parameters are the same as described for the M6800 MDOS Macro Assembler (Paragraph G.1).

1. The "Z" option does not exist.
2. The "W" option exists and indicates that warnings should be printed. "-W" suppresses warnings. The default is to print warnings.

3. If the "M" command line option is specified, warnings as well as error messages are directed to the line printer.

4. With 32K bytes of memory, the M6809 Macro Assembler can accommodate about 740 (decimal) symbols.

G.4 M6800 EDOS MACRO ASSEMBLER

The M6800 Macro Assembler is invoked from the EDOS command line, as are other EDOS commands. However, the RASM command requires that the system has a minimum of 16K bytes of memory. The format of the command line is:

```
RASM,[<list>],[<object>],<name 1>[,<name 2>,...,<name n>]
```

where <list> specifies whether or not a source listing is to be produced, <object> specifies whether or not an object file is to be produced, and <name i> (i=1 to n) are the names of EDOS source files. Each file name must be a valid EDOS file name (five characters). If multiple source files are specified, only the last file should contain an END directive. If an END directive is encountered prior to the last file, the assembly will not include files after the END directive.

The <list> can be either the line printer (#LP), the system console (#CN), an EDOS file name, or null (indicated by a comma only). If no <list> is specified, no source listing will be produced. If an EDOS file name is used to receive the source listing, then no object file can be created on the diskette at the same time.

The <object> can be either the line printer (#LP), the system console (#CN), an EDOS file name, or null (indicated by a comma only). If an EDOS file name is used to receive the object file, then no source listing can be created on the diskette at the same time. The line printer or system console should not be used if the program is being assembled with the relocatable option (OPT REL).

The EDOS Macro Assembler does not support the M6801 instruction set or the printing of sequence numbers on the left. If sequence numbers are in the source file, they will only be printed if the OPT SE option is in effect.

Each symbol in the symbol table requires ten bytes. Thus, if the minimum of 16K bytes of memory is used, the Macro Assembler can accommodate about 270 (decimal) symbols. However, if the cross reference option is used, the symbol table requirements differ. In this case, an additional ten bytes are required by each symbol for every four references to that symbol. If macro definitions are used (MACR directive), the available symbol table space will be smaller.

Following are examples of valid EDOS command lines used to invoke the Macro Assembler:

RASM,#CN,PROGO,PROGS

> This command line will cause the file PROGS to be assembled. A source listing will be produced on the system console. The object file PROGO will also be created on the diskette. Both source and object files are on drive zero.

RASM,,PROGO:1,PROGS

This command line will cause the file PROGS to be assembled. However, no source listing will be produced. The object file, PROGO, will be created on drive one.

RASM,#LP,,PROG1,PROG2,PROG3

This command line will cause the files PROG1, PROG2, and PROG3 to be assembled as if they were one contiguous source file. A source listing is produced on the system line printer. No object file will be created.

G.5 M6800/M6801 TAPE MACRO ASSEMBLER

The tape version of the Macro Assembler is loaded via EXbug. When the EXbug prompt:

EXbug V.R

is displayed, the command

LOAD

should be entered. EXbug will respond with the prompt:

SGL/CONT

to which the operator should respond with an "S". The tape should then proceed to be loaded into memory. EXbug will display its prompt again after the load has completed.

The Macro Assembler is given control via the command:

600;G

(either from MAID, if using EXbug version 1.1 or 1.2, or directly from EXbug, if using version 2.0). The Macro Assembler will then display a sign-on message, followed by the prompt:

```
#LIST,#OBJECT:
?
```

The operator must respond with the proper device designators as follows:

Designator	Device
#CN	Console printer
#CP	Console punch
#LP	Line printer
null	No output desired

For example, the operator response:

#CN,#CP

causes the source listing to be directed to the console printer, and the object file to be directed to the console punch. The operator response:

#LP

causes the source listing to be directed to the line printer, and no object file to be created. The operator response:

,#CN

causes no source listing to be generated, and an object file to be displayed on the console printer. A null response for both devices (carriage return only) will cause neither a source listing nor an object file to be created.

Next, the Macro Assembler will display the message:

```
SOURCE DEVICE:
?
```

to which the operator must enter the device designator that contains his source input file. The console reader (#CR) or the EXORtape (high-speed paper tape reader) (#HR) is the only valid designator for the source device. The source tape must be loaded and ready to be read before this response is given.

If an END directive is not encountered in the source file (i.e., a tape time-out occurred), then the assembler will redisplay the "SOURCE DEVICE" prompt, enabling the operator to load another source file. This process will continue until an END directive is encountered in a source file. If no source files contain an END directive, the operator can respond with the letter "E", followed by a carriage return to the "SOURCE DEVICE" prompt. This will end pass one of the assembler and will cause an error to be generated indicating that no END directive was encountered.

When the END directive is encountered, or when the "E" is entered by the operator as explained above, the assembler will end pass one, and begin pass two. This is indicated by the following display:

```
PASS 2
SOURCE DEVICE:
?
```

The operator must then reload all of the source tapes in the same sequence as they were loaded during the first pass. The specification of the device is the same as during pass one. The termination of pass two is also the same as during the first pass. During pass two, the source listing and the object file, if specified, will be produced.

After pass two is terminated, the assembler will display another question mark prompt (?) to indicate that it is ready to assemble another program. The source listing and object device designators should be entered at this point if another assembly is to occur.

If the operator detects an error in an input line that he has entered prior to depressing the terminating carriage return, the CTL-X keys can be depressed to cancel the entire line, allowing a new line to be input; or the CTL-H keys can be depressed causing the previously entered character to be deleted. The character deleted is redisplayed on the console as positive feedback that it was removed from the input line.

Each symbol in the symbol table requires ten bytes. Thus, if the minimum of 16K bytes of memory is used, the Macro Assembler can accommodate about 360 (decimal) symbols. However, if the cross reference option is specified, the symbol table requirements differ. In this case, an additional ten bytes are required by each symbol for every four references to that symbol. If macro definitions are used (MACR directive), the available symbol table space will be smaller.

The tape version of the Macro Assembler does not support the printing of sequence numbers on the left margin. If sequence numbers are contained in a file, they can only be printed with the OPT SE directive; then they will be printed in the right margin of the source listing. The tape version of the Macro Assembler does not support the relocatable option either. Thus, all directives dealing with program sections and relocatable features cannot be used.

APPENDIX H

SAMPLE PROGRAMS

The following example illustrates the various Macro Assembler directives that can be used in any program, regardless of whether or not it is assembled with the relocatable option. An attempt has been made to show all of the different types of constants and expression formats that can be used. Although the listing format shown is for the M6800 Macro Assembler, that is the only difference between that and the M6805 and M6809 Macro Assemblers for this example.

The comments contained in the example serve to document what the different directives are used for. Chapter 4 describes all of the directives in detail, and should be consulted for a description of each directive, if necessary.

```
PAGE  00  EXAMPL  .SA:0

00001                       *
00002                       * THIS EXAMPLE ILLUSTRATES THE USE OF THE VARIOUS
00003                       * ASSEMBLER DIRECTIVES THAT DO NOT INVOLVE
00004                       * PROGRAM RELOCATION.
00005                       *
00006                       *
00007                       *
00008                       * TURN ON OPTIONS TO PRINT SYMBOL TABLE AND TO
00009                       * GENERATE OBJECT LISTING FROM FCB, FDB, AND FCC
00010                       *
00011                              OPT    S,G
00012                       *
00013                       * USE DEFAULT VALUE OF PROGRAM COUNTER
00014                       * FOR INITIAL ORIGIN
00015                       *
00016                       *
00017                       * BSZ -- BLOCK STORAGE OF ZEROES
00018                       *        FIRST FORM USES SIMPLE CONSTANT
00019                       *        SECOND FORM USES COMPLEX EXPRESSION
00020                       *
00021A 0000     0005   A           BSZ    5          . FIVE BYTES
00022A 0005     0006   A LABELO BSZ    $10*2/2-$10+@77-76Q+101B . 6 BYTES
00023                       *
00024                       * EQU -- ASSIGN VALUE TO LABEL.  FIRST FORM USES
00025                       *        PROGRAM COUNTER IN EXPRESSION.  SECOND
00026                       *        FORM USES * AS BOTH PC AND MULTIPLY
00027                       *        OPERATOR.  THIRD AND FOURTH FORMS USE
00028                       *        SHIFT OPERATOR.
00029                       *
00030           000B   A TAG1   EQU    *          . USE OF PROGRAM COUNTER
```

```
00031             003C  A TAG2   EQU    ***/2     . CALC PC*PC/2
00032             0B00  A TAG3   EQU    TAG1!<8   . SHIFT LSB INTO MSB
00033             0B00  A TAG4   EQU    TAG1!<(2!^3) . SAME AS TAG3
00034                   *
00035                   * FCB -- FORM CONSTANT BYTE
00036                   *
00037A 000B       0C    A TAG5   FCB    12        . FORM A SINGLE BYTE
00038A 000C       0A    A        FCB    10,$10,&10,@10,%10,'1,'0,TAG3!>8,-1
     A 000D       10    A
     A 000E       0A    A
     A 000F       08    A
     A 0010       02    A
     A 0011       31    A
     A 0012       30    A
     A 0013       0B    A
     A 0014       FF    A
00039A 0015       0A    A        FCB    10,,,20   . USE OF NULL OPERANDS
     A 0016       00    A
     A 0017       00    A
     A 0018       14    A
00040                   *
00041                   * FDB -- FORM CONSTANT DOUBLE BYTE
00042                   *
00043A 0019       000C  A        FDB    12        . FORM A DOUBLE BYTE
00044A 001B       000A  A        FDB    10,$10,&10,@10,%10,'1,'0,TAG3!>8,-1
     A 001D       0010  A
     A 001F       000A  A
     A 0021       0008  A
     A 0023       0002  A
     A 0025       0031  A
     A 0027       0030  A
     A 0029       000B  A
     A 002B       FFFF  A
00045A 002D       000A  A        FDB    10,,,20   . USE OF NULL OPERANDS
     A 002F       0000  A
     A 0031       0000  A
     A 0033       0014  A
00046                   *
00047                   * FCC -- FORM CONSTANT CHARACTER STRING
00048                   *
00049A 0035       41    A        FCC    5,ABCDE   . STRING "ABCDE"
     A 0036       42    A
     A 0037       43    A
     A 0038       44    A
     A 0039       45    A
00050A 003A       41    A        FCC    5,A       . STRING "A    "
     A 003B       20    A
     A 003C       20    A
     A 003D       20    A
     A 003E       20    A
00051                   *
00052                   * TURN OF GENERATION OF OBJECT CODE LISTING FRO
00053                   *
00054                            OPT    NOG       .
```

```
00055                          *
00056A 003F     41       A STR_2   FCC     "ABC #$%&'() STRING"
00057A 0051     42       A STR$1   FCC     ABCDEFA   . STRING "BCDEF"
00058                          *
00059                          * REORIGIN THE PROGRAM COUNTER
00060                          *
00061A 0100                              ORG     $100      . PC=256 (DECIMAL)
00062                          *
00063                          * USE SPC DIRECTIVE TO SKIP 3 LINES
00064                          *



00066                          *
00067                          * RMB -- RESERVE MEMORY BYTES
00068                          *
00069A 0100     0005     A LOC.    RMB     5         . FIVE BYTES
00070                          *
00071                          * SET -- INITIALIZE TEMPORARY VALUE TO SYMBOL
00072                          *
00073           0001     A SKIP$1  SET     1         . CHANGEABLE SYMBOL
00074A 0105     0001     A         RMB     SKIP$1    . ONE BYTE
00075           0002     A SKIP$1  SET     SKIP$1+1  .
00076A 0106     0002     A         RMB     SKIP$1    . TWO BYTES
00077           0003     A SKIP$1  SET     SKIP$1+1  .
00078A 0108     0003     A         RMB     SKIP$1    . THREE BYTES
00079                          *
00080                          * END -- END OF PROGRAM
00081                          *
00082                              END
TOTAL ERRORS 00000--00000


LABELO 0005   LOC.    0100   SKIP$1 0003   STR$1   0051   STR_2   003F
TAG1   000B   TAG2    003C   TAG3   0B00   TAG4    0B00   TAG5    000B
```

H.1 M6800 PROGRAMS

The next two examples illustrate the use of the relocation scheme. The first program is a "main" program that calls a subroutine which is assembled external to the main program. The main program sets up the parameters prior to calling the subroutine. These two examples also show the format of the program listing, as well as the usage of the various addressing modes and relocatable directives. First, the main program is shown.

```
PAGE  001  RELMAIN .SA:1

00001                     *
00002                     * THIS EXAMPLE ILLUSTRATES THE USE OF THE
00003                     * RELOCATABLE DIRECTIVES.
00004                     *
00005                     *
00006                     *
00007                     * TURN ON RELOCATABLE AND CROSS REFERENCE
00008                     * TABLE OPTIONS
00009                     *
00010                            OPT     REL,CRE
00011                     *
00012                     * DEFINE THE EXTERNAL REFERENCES TO A
00013                     * MOVE CHARACTER SUBROUTINE.  "MOVE"
00014                     * IS THE ENTRY POINT TO
00015                     * THE ROUTINE; "FROM" IS A POINTER
00016                     * TO A SOURCE STRING; AND "TO" IS A
00017                     * POINTER TO A DESTINATION STRING.
00018                     *
00019                            XREF    BSCT:FROM,TO,PSCT:MOVE
00020                     *
00021                     * DEFINE ENTRY POINT INTO EXBUG
00022                     *
00023             FCF4  A EXBUG  EQU     $FCF4
00024                     *
00025                     * DEFINE A STRING, BUFFER, AND STACK
00026                     * IN THE DATA SECTION
00027                     *
00028D 0000                      DSCT
00029D 0000       001D  A        RMB     29         . STACK AREA
00030D 001D       0001  A STACK  RMB     1          . TOP OF STACK
00031D 001E       57    A STRING FCC     "WILL BE MOVED TO BUFFER"
00032             0035  D STREND EQU     *          . END OF STRING
00033D 0035       0050  A BUFFER RMB     80         . DESTINATION BUFFER
00034                     *
00035                     * DEFINE THE MAIN PROGRAM IN THE
00036                     * PROGRAM SECTION
00037                     *
00038P 0000                      PSCT
00039             0000  P START  EQU     *          .
00040P 0000 8E 001D  D           LDS     #STACK     . INITIALIZE STACK POINTER
```

```
00041P 0003 CE 001E   D         LDX     #STRING   .  SOURCE STRING
00042P 0006 DF 00     A         STX     FROM      .
00043P 0008 CE 0035   D         LDX     #BUFFER   .  DESTINATION AREA
00044P 000B DF 00     A         STX     TO        .
00045P 000D C6 17     A         LDAB    #STREND-STRING .  LENGTH TO MOVE
00046P 000F BD 0000   A         JSR     MOVE      .  ROUTINE TO MOVE
00047P 0012 7E FCF4   A         JMP     EXBUG     .  EXIT TO DEBUG MONITOR
00048                   *
00049           0000  P         END     START     .  STARTING EXECUTION ADDRE
TOTAL ERRORS 00000--00000
```

```
 D 0035 BUFFER 00033*00043
   FCF4 EXBUG  00023*00047
RB      FROM   00019*00042
RP      MOVE   00019*00046
 D 001D STACK  00030*00040
 P 0000 START  00039*00049
 D 0035 STREND 00032*00045
 D 001E STRING 00031*00041 00045
RB      TO     00019*00044
```

Next, the "MOVE" subroutine is shown.

```
PAGE  001  MOVE    .SA:1

00001                         *
00002                         * THIS EXAMPLE IS THE "MOVE" ROUTINE
00003                         * CALLED BY THE PREVIOUS EXAMPLE.
00004                         *
00005                         *
00006                                OPT    REL,CRE    .
00007                         *
00008                         * DEFINE THE EXTERNAL SYMBOLS
00009                         *
00010                                XDEF   MOVE,FROM,TO
00011                         *
00012                         * RESERVE SPACE IN DIRECT ADDRESSING AREA
00013                         * FOR THE SOURCE AND DESTINATION
00014                         * POINTERS.
00015                         *
00016B 0000                          BSCT
00017B 0000       0002    A FROM     RMB    2          . SOURCE POINTER
00018B 0002       0002    A TO       RMB    2          . DESTINATION POINTER
00019                         *
00020                         * DEFINE THE "MOVE" SUBROUTINE
00021                         * ENTERED WITH "B" = NO. BYTES IN SOURCE
00022                         * STRING.  "FROM" AND "TO" SET UP BY
00023                         * THE CALLING PROGRAM.
00024                         *
00025P 0000                          PSCT
00026             0000    P MOVE     EQU    *          . ENTRY POINT
00027P 0000 DE 00         B          LDX    FROM       . PICK UP SOURCE ADDRESS
00028P 0002 A6 00         A          LDAA   0,X        . GET SOURCE BYTE
00029P 0004 08                       INX               .
00030P 0005 DF 00         B          STX    FROM       . SAVE INCREMENTED POINTER
00031P 0007 DE 02         B          LDX    TO         .
00032P 0009 A7 00         A          STAA   0,X        . STORE DESTINATION BYTE
00033P 000B 08                       INX               .
00034P 000C DF 02         B          STX    TO         . SAVE INCREMENTED POINTER
00035P 000E 5A                       DECB              . DECREMENT COUNTER
00036P 000F 26 EF 0000               BNE    MOVE       . LOOP UNTIL ZERO .
00037P 0011 39                       RTS               . RETURN TO CALLER
00038                         *
00039                                END
TOTAL ERRORS 00000--00000


DB 0000 FROM      00010 00017*00027 00030
DP 0000 MOVE      00010 00026*00036
DB 0002 TO        00010 00018*00031 00034
```

H.2 M6805 PROGRAM

The following example illustrates the use of the bit instructions.

```
PAGE  001  H2         .SA:1

00010 00001                               *
00020 00002                               *TSTBIT CHECKS AN I/O BIT AND SETS
00030 00003                               *OR CLEARS SOME BIT FLAGS
00040 00004                               * DEPENDING ON STATE OF I/O BIT
00050 00005                               *
00060 00006             0004      A INPUT  EQU    $4        INPUT DATA
00070 00007A 0040                          ORG    $40
00080 00008A 0040       0001      A FLAG1  RMB    1         BIT FLAGS
00090 00009A 0041       0002      A FLAG2  RMB    2         BIT FLAGS
00100 00010A 0080                          ORG    $80
00110 00011             0080      A TSTBIT EQU    *
00120 00012A 0080 07 04 06 0089            BRCLR  3,INPUT,OFF
00130 00013                       *INPUT BIT IS ON -- SET SOME BIT FLAGS
00140 00014A 0083 1A 40           A        BSET   5,FLAG1
00150 00015A 0085 14 40           A        BSET   2,FLAG1
00160 00016A 0087 20 06     008F           BRA    CONT
00170 00017                       *INPUT BIT IS OFF -- CLEAR SOME BIT FLAG
00180 00018A 0089 11 41           A OFF    BCLR   0,FLAG2
00190 00019A 008B 1D 41           A        BCLR   6,FLAG2
00200 00020A 008D 1F 41           A        BCLR   7,FLAG2
00210 00021             008F      A CONT   EQU    *         CONTINUE PROCESSI
00220 00022                                END
TOTAL ERRORS 00000--00000
```

H.3 M6809 PROGRAMS

The following example illustrates how a program can take advantage of the direct addressing mode without being a relocatable program using BSCT.

```
00001                              *
00002                              *THIS PROGRAM HANDLES AN INTERRUPT FROM
00003                              *     AN INPUT DEVICE--IT GETS CONTROL ON
00004                              *     AN IRQ FROM A PIA, INPUTS A CHAR,
00005                              *     CLEARS THE INTERRUPT, PUTS THE CHAR
00006                              *     IN A BUFFER, INCREMENTS THE BUFFER
00007                              *     PTR, TESTS FOR END OF LINE, RESTORES
00008                              *     REGISTERS, AND RETURNS
00009                              *
00010A 2000                               ORG    $2000
00011            000D      A EOL    EQU    $D        CR IS END OF LINE IND.
00012A 2000      00        A MODEM  FCB    0
00013A 2001      2003      A BUFPTR FDB    BUF
00014A 2003      0064      A BUF    RMB    100
00015                              *SET UP DP PSEUDO REG. FOR ASSEMBLER
00016            0020      A        SETDP  $20
00017                              *SET UP DP REGISTER FOR EXECUTION
00018A 2067 86   20        A        LDA    #$20
00019A 2069 1F   8B        A        TFR    A,DP
00020A 206B 96   00        A        LDA    MODEM     CLEARS PIA IRQ
00021A 206D 9E   01        A        LDX    BUFPTR    GET PTR
00022A 206F A7   80        A        STA    ,X+       STORE CHAR
00023A 2071 9F   01        A        STX    BUFPTR    UPDATE PTR
00024A 2073 81   0D        A        CMPA   #EOL      END OF LINE?
00025A 2075 27   01   2078          BEQ    EOLGP     IF YES, MORE TO DO
00026A 2077 3B                      RTI              ELSE, RETURN
00027A 2078 20   FE   2078 EOLGP    BRA    *
00028                               END
TOTAL ERRORS 00000--00000
TOTAL WARNINGS 00000--00000
```

The following example illustrates how position independent code can be generated by using the PCR indexing mode.

```
00001                              *
00002                              *SUBSEQ SUBTRACTS A SEQUENCE OF DECIMAL
00003                              *      DIGITS (IY) FROM ANOTHER SEQUENCE
00004                              *      OF DECIMAL DIGITS (IX) AND STORES
00005                              *      THE RESULT (US)
00006                              *      ALL STRINGS ARE COUNT BYTES LONG

00008A 0000      99           A MINUEN FCB    $99,$99,$99,$99,$99
00009A 0005      99           A        FCB    $99,$09,$00,$00,$00
00010A 000A      01           A SUBTRA FCB    $01,$09,$00,$00,$00
00011A 000F      99           A        FCB    $99,$00,$54,$32,$11
00012A 0014      000A         A RESULT RMB    10
00013            000A         A COUNT  EQU    10

00015A 001E 30   8C E9                 LEAX   MINUEN+COUNT,PCR
00016A 0021 31   8C F0                 LEAY   SUBTRA+COUNT,PCR
00017A 0024 33   8C F7                 LEAU   RESULT+COUNT,PCR
00018A 0027 C6   0A           A        LDB    #COUNT
00019A 0029 8D   02     002D           BSR    SUBSEQ
00020A 002B 20   FE     002B           BRA    *
00021                              *
00022A 002D 1A   01              SUBSEQ SEC                SET CARRY
00023A 002F 34   01           A        PSHS   CC          CARRY TEMP
00024A 0031 86   99           A LOOPS  LDA    #$99        THE TEN'S COMPLEMENT
00025A 0033 A0   A2           A        SUBA   0,-Y        NO CARRY POSSIBLE
00026A 0035 35   01           A        PULS   CC          THE SAVED CARRY
00027A 0037 A9   82           A        ADCA   0,-X        DO A BINARY ADD
00028A 0039 19                         DAA                BACK TO BCD
00029A 003A 34   01           A        PSHS   CC          SAVE THE CARRY
00030A 003C A7   C2           A        STA    0,-U        STORE THE RESULT
00031A 003E 5A                         DECB               DONE?
00032A 003F 26   F0     0031           BNE    LOOPS       IF NOT, GO AGAIN
00033A 0041 35   81           A        PULS   CC,PC       CLEAN UP STACK AND RET
00034                                  END
TOTAL ERRORS 00000--00000
TOTAL WARNINGS 00000--00000
```