6800/6809 ASSEMBLER

V1.4

USER'S MANUAL

SOFTWARE DYNAMICS

2111 W. Crescent, Suite G, ▲ Anaheim, CA 92801

ASM 1.4

REFERENCE MANUAL

7th Printing

TABLE OF CONTENTS

SOFTWARE DYNAMICS
ASM -- 68ØØ/68Ø9 MICROPROCESSOR ASSEMBLY LANGUAGE

INTRODUCTION

ASM -- A SMART ASSEMBLER ON A LITTLE MACHINE

ASM is a sophisticated family of assemblers for 68ØØ, 68Ø1 and 68Ø9 microprocessors, and is intended for operation under the SDOS operating system.

There are four such assemblers:

ASM68ØØ.68Ø, assembling 68ØØ/68Ø1 code and executing on a 68ØØ system
ASM68ØØ.689, assembling 68ØØ/68Ø1 code and executing on a 68Ø9 system
ASM68Ø9.68Ø, assembling 68Ø9 code and executing on a 68ØØ system
ASM68Ø9.689, assembling 68Ø9 code and executing on a 68Ø9 system

These assemblers are powerful, highly flexible development tools. They are fast. Most important, they are designed by professionals with the professional user in mind.

Significant features include:

--Multi-level conditional assembly. ASM's facilities are unrivalled in the industry.

--Multi-level INCLUDE files.

--Extensive listing format control.

--Big-assembler computational power due to extensive operator set.

--Big-assembler performance due to hash-coded symbol table lookup.

--Symbol table dumps sorted by name and by value; unused symbols are flagged.

--Big-assembler error diagnosis. Readable error messages are produced on the listing; error lines are always listed; and a special summary at the end of the assembly tells you which lines had errors.

--68ØC coding style allows code to work on 68ØØ, 68Ø1 or 68Ø9 by simply reassembling.

NOTATION DEFINITION

When introducing a construct or command, the following notation will be used to describe its allowable syntax.

--Graphic characters (e.g., +, /, -) and symbols printed in upper case denote strings of characters which must be present exactly as written in the manual. Example:

```
      ,X
```

denotes the string ",X".

--Symbols written in angle brackets denote a class of possible inputs. For example,

```
      <NUMBER>
```

denotes any string which conforms to ASM's definition of a number. Hence, the string

```
      345
```

would qualify.

--Curly brackets are used to denote that a certain item is optional.

```
      {,X}
```

indicates that the string ",X" may occur, but need not.

Ranges of possibilities of which one must be chosen are denoted by listing the alternatives vertically.

```
      X
      ,X
      <EXP>,X
```

indicates that either "X", ",X", or an expression followed by ",X" must be present. Note that this could also have been written as

```
      X
      {<EXP>},X
```

or

```
      { {<EXP>} , } X
```

--Finally, ellipsis ("...") are used to indicate the possibility of indefinite repetition when embedded in text. For example,

```
      <EXP> { ,<EXP> ... }
```

indicates that a series of one or more expressions separated by commas is acceptable. Ellipsis on a line by itself represents some indefinite sequence of source lines. For example:

```
      line 1
      ...
      line n
```

indicates that some sequence of lines exists between line 1 and line n.

GENERAL DISCUSSION

All assemblers in the ASM family have all the properties described in this manual, unless specifically indicated. For the most part, differences between the assemblers are limited to the instruction set intended for assembly by ASM.

This manual assumes that the reader knows the M68ØØ/M68Ø1 and/or the M68Ø9 machine instruction set. It is a manual on the use of the assembler. It is not, nor is it intended to be, a tutorial on the instruction set for these microprocessors.

To use ASM, the programmer first prepares a "source" disk file. This can be done by using EDIT or SEDIT under SDOS. Then ASM is invoked, and given the name of the source file to be assembled, and names of listing and binary object files to be produced.

ASM is a two-pass assembler. That is, the entire source file is read twice. On the first pass through the source file, all assembly errors are suppressed, and certain operations (like producing a listing or binary) are not performed. This pass is performed primarily to assign symbols their values.

At the end of the first pass, all symbols should be defined; ASM rewinds the source file, and processes it again. This time it produces a listing file and a binary file, according to the user's directions.

SOURCE FILE FORMAT

ASM places very few restrictions on the format of the source file. Its only requirements are:

1) Source lines must be terminated with carriage returns.

2) Nulls may not be significant characters in the file; they will always be ignored.

3) If the file is divided up into "forms" using form-feed characters, the form-feeds must immediately follow a carriage-return.

All other ASCII characters are legal and will be printed in the listing without complaint. Control characters (with the exception of tab) will always be printed as "^" followed by an appropriate letter. For example, control-A will be listed as "^A".

The tab character is treated somewhat differently. If it appears as part of a character string, it will be listed as "^I"; otherwise it will cause one or more blanks to be output, until the print carriage is positioned at a tab stop or is past the last tab stop. Tabs may be adjusted by the user via the TABS directive.

A form-feed character that does not follow a carriage return will always be treated like any other ASCII control character.

LINE FORMAT

ASM has a free-format line syntax. Fields may begin in any column and are separated from one another (or "delimited") by blanks or tabs. Each field may be broken into subfields, each subfield being separated from the next by commas.

Some consequences of ASM's line syntax are:

--Blank lines may be freely inserted to format the listirg as desired.

--Comment lines may be specified either with a star ("*") in column 1 or with a series of blanks, a semicolon and the comment. Examples:

```
* THIS IS A COMMENT LINE.  THE "*" IS IN COLUMN 1.
; THIS IS ALSO A COMMENT LINE.
  ; THIS IS A COMMENT LINE, TOO.
          ; ANY NUMBER OF BLANKS CAN PRECEDE ";".
```

--Blank or tab characters not embedded in a string ALWAYS terminate a field. In particular, this means that a comma should not usually be followed by a blank, as one would ordinarily do when typing. Also, this means that blanks may not be inserted in the middle of an expression.

--Consecutive blanks or tabs are treated as a single blank.

A line consists of the following parts, any of which may be absent. The scanning of a line is always stopped by a ";" (semi-colon) character encountered outside a text string.

1) LABEL FIELD (or LF). The label field, if non-null, must begin in the first column of the line. (see "Line Numbers", below for exception). This field is generally used to specify symbols whose values are to be changed or determined by the command given on the same line. If the label field begins with an asterisk, the entire line is treated as a comment. It will be listed but otherwise ignored.

2) COMMAND FIELD (or CF). The command field is the second field of the line. This field is examined by ASM in order to determine how to process the line. If the command field is absent, ASM will treat the line as an implicit EQU * if a label is present, otherwise the line will be treated as a comment.

3) ARGUMENT FIELD (or AF). The argument field is the third field of the line. It contains all arguments needed by the command. For an argument field to be present, there must also be a command field. Example:

```
LDAA    AF
|       |
|       > ARGUMENT FIELD
> COMMAND FIELD
```

4) COMMENT FIELD. Any portion of the line which is not scanned for a given command, or which is separated from the rest of the line by a semicolon, is treated as commentary. It is listed but otherwise ignored (but note that this is not the case when the semicolon is part of or delimits a string. See discussions of FCC, TITLE directives.)

SOURCE LINE NUMBERS

ASM can accept files containing line numbers at the beginning of each line, as long as the user alerts ASM via the "WITH" directive.

If line numbers are present in the file, the line format is modified somewhat. Line numbers are handled in the following fashion.

1) Any string of zero or more digits or periods which starts a line will be listed but otherwise ignored.

2) If the next character is a blank or a tab, it will also be ignored.

3) The next character in the file will be the first significant character of the line.

4) No error checking is done with respect to line numbers.

Examples: (Note that <TAB> represents the ASCII tab character. Assume that ASM has been directed to expect line numbers.)

```
409.17 <TAB> LABEL <TAB> LDAA <TAB> FROG
\----------/ \---/        \--/       \--/
   LINE #     LF           CF         AF

409.17LABEL <TAB> LDAA <TAB> FROG
\----/\---/        \--/       \--/
 LN #  LF           CF         AF
```

Because of the definition of the line number format, rather unexpected things can happen:

```
409.17 <TAB> LDAA <TAB> FROG
\----------/ \--/        \--/
   LINE #     LF          CF
```

In the above example, the LDAA is erroneously treated as a label. The proper way to enter the line is:

```
409.17 <TAB> <TAB> LDAA <TAB> FROG
\----------/   |    \--/       \--/
   LINE #      |     CF         AF
             (NO LF)
```

Note that the line number can be absent:

```
<TAB> LABEL <TAB> LDAA <TAB> FROG
\---/ \---/        \--/       \--/
 LN #  LF           CF         AF
```

The moral is, be careful with line numbers.

EXPRESSIONS

ASM provides the user with the ability to perform sophisticated computations at assembly time. ASM expression handling represents a significant step beyond the capabilities provided by Motorola-standard assemblers. Many more operators are handled, as well as parenthetical and heirarchical grouping of subexpressions.

These extensions do not represent a burned bridge between ASM and Motorola standard assemblers; in fact, ASM can be directed via the "WITH" command to supress heirarchical considerations and evaluate operators from left to right; thus sources currently in development using Motorola-standard assemblers can be shifted to ASM without difficulty.

VALUES

Numeric values can be specified in expressions in two ways: manifestly and implicitly. All values share certain attributes:

--They are all sixteen-bit quantities.

--They are usually considered as two's complement signed numbers, where the most-significant bit is the sign of the number: positive if reset, negative if set.

--Certain operators treat the values as 16-bit unsigned quantities, in particular the multiply and divide group.

MANIFEST VALUES

"Manifest values" are those values which are entirely self-defining; that is, the text of the manifest value completely specifies the value to be used. This section describes the various manifest constants and their variations.

DECIMAL NUMBERS

The simplest manifest value is a decimal number. The value is represented by a string of decimal digits, in ordinary base-1Ø notation.

```
     <DIGIT> { <DIGIT> ... }
```

If the value specified exceeds 65535, "Overflow" is reported. If a letter is detected in the middle of a number, "Illegal Digit" is reported.

Examples:

```
        346
        ØØØ443
        ØØØØØØØ1     (Many Leading Zeroes is OK)
        1ØØØØØØØ     (But this is too big -- "Overflow" is reported)
        1139A4       (Also illegal -- bad digit)
```

BINARY NUMBERS

Numbers may be specified in bases other than base ten. Binary (base two) numbers are represented as follows:

```
        % <B-DIGIT> { <B-DIGIT> ... }
        <B-DIGIT> { <B-DIGIT> ... }   B
```

<B-DIGIT> must be either 'Ø' or '1'. If a digit outside this range is seen, "Illegal Digit" is reported. If the assembled value exceeds 65535, "Overflow" is reported.

Examples:

```
        %1Ø11Ø                  (Binary repesentation of 22 base 1Ø)
        1Ø11ØB                  (Same thing using suffixed form)
        %111ØØØ111ØØØ111ØØØ     (Too big -- causes "overflow")
        Ø1Ø21ØB                 (Illegal -- "2" is not a binary digit)
```

Lower case qualifiers are also accepted:

```
        %1Ø11b                  (Same as %1Ø11B)
```

OCTAL NUMBERS

Numbers may be specified in base eight, "octal notation." Such numbers have the form:

```
        @ <O-DIGIT> { <O-DIGIT> ... }
        <O-DIGIT> { <O-DIGIT> ... }  O  (The letter "oh")
        <O-DIGIT> { <O-DIGIT> ... }  Q
```

<O-DIGIT> may be any of the digits "Ø" through "7"; if "8" or "9" are seen, "Illegal Digit" is reported. If the value exceeds 65535, "Overflow" is reported.

Examples:

```
        @26          (Octal representation of 22, base ten)
        26O          (The same, using suffix form)
        26Q          (Alternate suffix form)
        @759         (Illegal -- '9' is not an octal digit)
        @3777777     (Too big -- "overflow" is reported)
```

Note that lower case qualifiers are also accepted:

```
        Ø26q
        73o
```

HEXADECIMAL NUMBERS

Numbers may be specified in base 16, so-called "hexadecimal notation." Each digit represents a number from Ø to 15; the digits 1Ø through 15 are represented by the letters "A" through "F". There are two forms:

```
        $ <HEXDIGIT> { <HEXDIGIT> ... }
        <DIGIT> { <HEXDIGIT> ... } H
```

A suffix-form hexadecimal value MAY NOT begin with one of the hex digits "A" through "F". Numbers whose first significant digit is "A" through "F" must have a leading zero (see examples).

If a letter that is not between "A" and "F" is seen, "Illegal Digit" is reported. If the value exceeds $FFFF, "Overflow" is reported.

Examples:

$16	(Hex representation of 22, base 1Ø)
16H	(Suffix form of same number)
ØAEH	(Suffix form of $AE)
AEH	(Invalid -- this is a symbol)
$EG	(Invalid -- "G" is not a hex digit)
$FFFE3	(Invalid -- too big for 16 bits. Overflow reported)

Lower case digits and qualifiers are accepted:

```
        Øaeh
        $f7f
```

CHARACTER VALUES

A value may be specified in terms of the ASCII code of an input character. Such values have the following form:

```
        ' <CHARACTER>
```

The "'" is a single quote character, ASCII $27. <CHARACTER> may be any character except carriage-return (subject to the restrictions mentioned in the section on "Source File Format"). If the character is a tab, it will be listed as "^I" for improved readability; hence the only time one will see "'" followed by a blank in a listing is when the quoted character actually is a blank.

The value used will be the 7-bit ASCII value of the character, with zeroes extended to fill the 9 most-significant bits of the value.

Examples:

```
        'A            (Specifies the value 65, base 1Ø)
        '             (Specifies the value 32, i.e. quoted blank)
        '^I           (Specifies the value 9)
        '<CR>         (Invalid -- "illegal string" is reported)
```

IMPLICIT VALUES

Obviously, manifest values would not be very useful by themselves. An important feature of any assembler is the ability to represent values symbolically. The user thus gains the ability to manipulate values without having to know what the actual values are.

THE "*" VALUE

ASM provides a special kind of implicit value -- the address that the next byte stored into memory will occupy. This value is called the LOCATION COUNTER. Whenever a star ("*") takes the place of a value in an expression, ASM substitutes the current value of the location counter. This value can be used like any other value; it is never a forward reference. At the beginning of each pass, "*" has the value zero.

"*" does not neccessarily remain the same throughout the processing of a statement (for example, see FDB directive). But on most statements, particularly all machine language instructions, it remains constant until all expressions on that line have been evaluated.

THE "*'" VALUE

ASM for 68Ø9 provides a second type of instruction counter, the location of the start of the next instruction. Whenever the star-apostrophe appears ("*'") within an instruction operand, it will yield the value of the following instruction's opcode byte. If the star-apostrophe appears in a non-instruction line, its value is identical with star ("*").

Examples:

```
                FDB     *'      ;Same as *
                #*'             ;Same as *
        A       SET     *'      ;Same as *
```

The reason for this special implicit value is it represents the PC (program counter) value that is used in PC relative addressing modes.

Example:

```
                LDA     QQQ-*',PC       ;Relative reference to loc QQQ
```

SYMBOLS

Symbols in ASM consist of a letter, a colon (":") or a period ("."), followed by a string of letters, digits, colons, periods, dollar signs ("$"), percent signs ("%") or at signs ("@"). A symbol may be of any length; however only the first thirty-two are used to distinguish one symbol from another.

Lower-case letters which appear in symbols are considered by ASM to be the same character as their upper-case equivalent. They will appear in lower case on the listing, but will match upper-case versions of the same letters.

The following are symbols:

```
        QRS
        qrs     (The same as QRS -- lower-case matches upper-case)
        L34
        A
        .BEGIN.WITH.DOT.LONGER.THAN.THIRTY
        .BEGIN.WITH.DOT.LONGER.THAN.THIRTY.TWO
                (ASM treats the last two as the same symbol)
        .BEGIN.WITH.DOT.LONGER.THAN.FIFTEEN
                (This symbol is different from both the above)
```

The following are NOT symbols:

```
        $DOLLAR (Does not begin with letter, colon or period)
```

RESERVED SYMBOLS

ASM has no "reserved symbols", however, some symbols have special interpretation in certain contexts. For 68ØØ/68Ø1 ASM, these symbols are "A", "B", "D", "S", and "X". For 68Ø9 ASM the symbols "A", "B", "D", "X", "Y", "S", "U", "CC", "DP", "PC" and "PCR" can have special interpretations. (See "Machine Instruction Lines" for details.)

PRE-DEFINED SYMBOLS

ASM has three pre-defined symbols:

M68ØØ
M68Ø1
M68Ø9

The purpose of these symbols is to support the 68ØC "concept", permitting conditional assembly dependent on the target CPU. For more information, see "Appendix D".

For the ASM68ØØ with 68Ø1 option disabled (no use of WITH M68Ø1 directive), the symbol M68ØØ has the value 1, and M68Ø1 and M68Ø9 have the value Ø. For ASM68ØØ with 68Ø1 option enabled (WITH M68Ø1 directive used), the symbol M68Ø1 has the value 1 and M68ØØ and M68Ø9 have the value Ø. ASM68Ø9 has M68Ø9 equal to 1 with M68ØØ and M68Ø1 equal to Ø.

FORWARD REFERENCES

A symbol may with some restrictions be used before it is defined. Such a use is called a FORWARD REFERENCE. In general, a forward reference may be used in any context that will allow ASM to allocate the same number of bytes at the same locations in pass one and pass two.

See Appendix A for a list of ASM directives which do not allow forward references.

COMPUTATION

Manifest and implicit values can be combined and operated upon at assembly time to produce new values. This is done by means of operators. ASM provides an extensive range of computational functions. These functions are divided into two categories: Monadic operators, which operate on one value to produce a result; and Dyadic Operators, which combine two values to produce a third.

MONADIC OPERATORS

```
     +<A>
```
Monadic plus does nothing. Its usual use is for clarity.
```
     +1      ==>     1
     +Ø      ==>     Ø
```

```
     -<A>
```
Monadic minus computes the two's complement of its argument.
```
     -1      ==>     $FFFF
     -Ø      ==>     Ø
```

```
      +\<A>
```
Monadic backslash computes the one's complement of its argument.
```
     \1      ==>     $FFFE
     \Ø      ==>     $FFFF
```

```
     &<A>
```
Monadic ampersand computes the logical inverse of its argument. ASM considers any zero or negative value to be logically "false"; any positive non-zero value is "true".
```
     &1      ==>     Ø
     &Ø      ==>     1
     &17     ==>     Ø
     &(-1)   ==>     1
```

DYADIC OPERATORS

<A>+<B>

Dyadic plus computes the arithmetic sum of its two arguments. No check is made for overflow.

```
     3+2               ==>      5
     $FAØØ+$1ØØ        ==>      $FBØØ
     $1Ø5+$FFFF        ==>      $1Ø4     (No overflow is reported)
```

<A>-<B>

Dyadic minus computes the two's complement arithmetic difference of its arguments. No overflow check is made.

```
     3-2               ==>      1
     $FAØØ-$1ØØ        ==>      $F9ØØ
     $1Ø5-$FFFF        ==>      $1Ø6
```

<A>*<B>

Dyadic star multiplies its arguments together as sixteen-bit unsigned quantities. If the result cannot be represented as a sixteen bit unsigned number, "Overflow" is reported.

```
     3*2               ==>      6
     $7ØØØ*2           ==>      $EØØØ    (No overflow)
     $FEØØ*2           ==>      $FCØØ    (Overflow)
```

<A>/<B>

Slash computes the quotient of <A> divided by <B>. The arguments are treated as sixteen-bit, unsigned quantities. If <B> is zero, "Overflow" is reported, and the result is zero.

```
     3/2               ==>      1        (Integer division)
     3Ø7/5             ==>      61
     $FEØØ/$1ØØ        ==>      $FE
     $17ØC/Ø           ==>      Ø        (Overflow)
```

<A>//<B>

Doubleslash computes the Covered Quotient of <A> and <B>. Covered Quotient is defined as (<A>+<B>-1)/<B>. This operator is useful in computing the number of <B>-byte units needed to hold <A> bytes. For example, if PROGSIZE is the number of bytes in a program, and SECTSIZE is the number of bytes in a disk sector, PROGSIZE//SECTSIZE is the number of disk sectors required to store that program.

```
3//2            ==>     2
307//5          ==>     62
$FE00//$100     ==>     $FE
$170C//0        ==>     0       (Overflow)
$FF03//$100     ==>     $100    (Note that this works even
                                though $FF03+$100-1 exceeds
                                16 bits)
```

<A>\<B>

Dyadic backslash computes the remainder of <A> divided by <B>.

```
3\2             ==>     1
307\5           ==>     2
$FE00\$100      ==>     0
$170C\0         ==>     0       (Overflow)
$FF03\$100      ==>     3
```

<A>##<B>

Doublehash performs a logical-shift operation. If <B> is positive, the result is <A> shifted left <B> bit places, with zeroes shifted into the least-significant bits. If <B> is negative, the result is <A> shifted right <B> bit places, with zeroes shifted into the most-significant bits. If <B> is zero, the result is <A>.

```
3##2            ==>     $C
$FE##8          ==>     $FE00
$FE00##-8       ==>     $FE
12345##16       ==>     0
```

<A>!<B>

Bang computes the bitwise logical-inclusive-or of its arguments.

```
2!1     ==>     3
0!4     ==>     4
7!2     ==>     7
```

```
     <A>&<B>
Dyadic  ampersand  computes  the  bitwise  logical-and  of  its
arguments.
     2&1       ==>      Ø
     Ø&4       ==>      Ø
     7&2       ==>      2
```

```
     <A>!!<B>
Doublebang  computes  the  bitwise  logical-exclusive-or  of  its
arguments.  It is defined as (<A>&\<B>)!(<B>&\<A>).
     2!!1      ==>      3
     Ø!!4      ==>      4
     7!!2      ==>      5
```

```
     <A>=<B>
Equal compares its arguments  and  returns 1 if they are equal, Ø
otherwise.
     3=3       ==>      1
     3=4       ==>      Ø
```

```
     <A>#<B>
Hash compares its arguments and  returns  Ø  if they are equal, 1
otherwise.
     3#3       ==>      Ø
     3#4       ==>      1
     <A> > <B>
```

Greater compares its arguments as signed numbers; returns 1 if left argument is greater than the right argument, Ø otherwise.

```
     3>3      ==>     Ø
     3>4      ==>     Ø
     4>3      ==>     1
     $FFFE>Ø  ==>     Ø
     Ø>$FFFE  ==>     1
```

(Note that $FFFE is interpreted here as -2.)

```
     <A> >= <B>
     <A> => <B>
```

Greater-equal is like Greater, but returns 1 if <A> is greater than or equal to <B>.

```
     3>=3             ==>     1
     3>=4             ==>     Ø
     4>=3             ==>     1
     $FFFE>=Ø         ==>     Ø
     Ø>=$FFFE         ==>     1
```

($FFFE is interpreted here as -2.)

```
     <A> < <B>
```

Less is like Greater, but returns 1 if <A> is less than <B>, Ø otherwise.

```
     3<3              ==>     Ø
     3<4              ==>     1
     4<3              ==>     Ø
     $FFFE<Ø          ==>     1
     Ø<$FFFE          ==>     Ø
```

($FFFE is interpreted here as -2.)

```
     <A> <= <B>
     <A> =< <B>
```

Less-equal is like Greater, but returns 1 if <A> is less than or equal to <B>, Ø otherwise.

```
     3<=3             ==>     1
     3<=4             ==>     1
     4<=3             ==>     Ø
     $FFFE<=Ø         ==>     1
     Ø<=$FFFE         ==>     Ø
```

($FFFE is interpreted here as -2.)

```
     <A> >> <B>
```

Logical-greater compares its arguments as UNSIGNED numbers; it returns 1 if <A> is greater than <B>, otherwise Ø.

```
     3>>3           ==>     Ø
     3>>4           ==>     Ø
     4>>3           ==>     1
     $FFFE>>Ø       ==>     1
     Ø>>$FFFE       ==>     Ø
```

($FFFE is interpreted here as 65534.)

```
     <A> >/ <B>
```

Logical-greater-equal is like Logical-greater, but returns 1 if <A> is greater than or equal to <B>, Ø otherwise.

```
     3>/3           ==>     1
     3>/4           ==>     Ø
     4>/3           ==>     1
     $FFFE>/Ø       ==>     1
     Ø>/$FFFE       ==>     Ø
```

($FFFE is interpreted here as 65534.)

```
     <A> << <B>
```

Logical-less is like Logical-greater, but returns 1 if <A> is less than <B>, otherwise Ø.

```
     3<<3           ==>     Ø
     3<<4           ==>     1
     4<<3           ==>     Ø
     $FFFE<<Ø       ==>     Ø
     Ø<<$FFFE       ==>     1
```

($FFFE is interpreted here as 65534.)

```
     <A> \< <B>
```

Logical-less-equal is like Logical-greater, but returns 1 if <A> is less than or equal to <B>, otherwise Ø.

```
     3\<3           ==>     1
     3\<4           ==>     1
     4\<3           ==>     Ø
     $FFFE\<Ø       ==>     Ø
     Ø\<$FFFE       ==>     1
```

($FFFE is interpreted here as 65534.)

COMPLEX EXPRESSIONS

ASM allows the user to create complex expressions involving many operators. In such expressions, the problem arises of order of computation. ASM provides two methods of specifying this order.

OPERATOR HIERARCHY

Unless otherwise instructed (via the "WITH" directive or the use of parentheses), ASM evaluates expressions using an operator-precedence algorithm; the order of evaluation, though generally left-to-right, can be modified according to which operators are used. In order to determine whether an operator gets to be evaluated, ASM looks at the operators which follow that operator in the expression. If the next operator gets to go first, ASM performs that operator and uses the result as the right argument to this operator -- but not before checking the operator-after-next, and so on. The ordering decision is made as follows:

1) The first operators performed will be monadic operators, from right to left.

```
     --SYM    <==>    -(-SYM)
     -\4      <==>     -(\4).
```

2) The next operator performed will be double-hash (shift).

```
     A##-3    <==>    A##(-3)
```

3) The next operators performed will be *, /, // and \, going from left to right across the expression.

```
     A*B/5    <==>    (A*B)/5
     A/5##4   <==>    A/(5##4)
     A/B*2    <==>    (A/B)*2
```

4) The next operators performed will be dyadic + and -, again from left to right.

```
     A+3-B    <==>    (A+3)-B
     A-3+B    <==>    (A-3)+B
     A-B-C    <==>    (A-B)-C
     A-B*4    <==>    A-(B*4)
```

5) The next operators evaluated will be the relational operators: =, #, <, <=, =<, >, >=, =>, <<, \<, >>, and >/.

```
     3*A<B            <==>    (3*A)<B
     4##2<<5/A+2      <==>    (4##2)<<((5/A)+2)
     B>3*A            <==>    B>(3*A)
     A<B<C            <==>    (A<B)<C
```

6) Last come the logical operators: !, &, and !!.

```
     A<B&C=D+2        <==>    (A<B)&(C=(D+2))
     A>B!C#D!!D<Ø     <==>    ((A>B)!(C#D))!!(D<Ø)
```

PARENTHESES

As implied in the above descriptions, parentheses can be used to specify explicitly the order of evaluation of a given expression. Parentheses must always be matched; that is, for every left parenthesis there must be one and only one corresspondng right parenthesis. A failure here will result in a "Syntax Error" for the line containing the expression.

Parentheses may be nested to any level.

They may be used even if WITH MCM has been specified.

LINE PROCESSING

ASM follows these rules when processing a line:

1) If the first character of the Label field is a star ("*"), the line is treated as a comment. It is listed but otherwise ignored.

```
      *       This is a comment
      *       Ignore me.
```

2) If the command field consists of a single symbol it is looked up in the list of opcodes (see sections on "Machine Instruction Lines" for lists of opcodes accepted). If found, the line is treated as a machine instruction line.

```
              LDX     3
              TSX             All of these
              CLV             are machine instruction lines
              PUL A           by rule (2).
```

3) If the command field is not an opcode symbol, then the symbol is checked in the list of ASM directives. If found, the line is treated as a directive line.

```
      J       EQU     17      This is a directive by rule (4)
              PAGE            So is this
              ORG     $47     As is this
```

4) Any line which does not qualify under rules (2), or (3) is processed as an implicit data statement (see below).

```
              LDAC    #34     This is an implicit data statement
              3,4*17          So is this
              (TSX)           So is this
```

6800/6801 MACHINE INSTRUCTION LINES

The major function of an assembler is the translation of symbolically specified machine instructions into a form directly understandable by the target computer. This section describes instructions translated by ASM6800, and presumes knowledge of the 6800/6801 instruction set, which can be found in the Motorola MC6801 Programming Manual. ASM is fully compatible with the Motorola 6800 standard assembler syntax, with several useful extensions.

--6801 opcodes are assembled if enabled by use of a WITH M6801 directive. 6800 equivalent instructions are assembled for 6801 instructions if a WITH M6801 directive is not given.

--Many 6809 assembly mnemonics and address modes are supported. This considerably simplifies construction of programs that will run on either processor, depending on which assembler is used. See section on M680C.

--Special set of opcodes to allow coding to work on 6800, 6801 and 6809. These opcodes are called 680C opcodes (See Appendix D). Many "convenience" instructions are included in 680C, which act as logical extensions to the '00/'09 instruction set, such as double register shifts, 16 bit memory increments, and decrements, etc.

--Automatic long branching. Short branches that are out-of-range will be assembled as a branch on complementary conditions around a JMP to the destination if the destination expression contains no forward references.

--Multiple labels are allowed, separated by commas. All labels present on an opcode line will be equated to the location counter, "*", before the rest of the line is processed.

--Two new inherent-addressing mode opcodes are defined, SK1 and SK2 (Skip 1 and Skip 2). These opcodes can be used to cause the CPU to skip one or two bytes before executing the next instruction, changing only the condition code flags. These opcodes will work on all standard 6800/6801 CPUs; they are really the opcode bytes of Bit A, Immediate and Compare X, Immediate instructions. Note that use of these instructions is NOT generally portable from 6800 to 6809 and vice versa.

--A, B, D, S, and X may be used as ordinary symbols as long as they are distinguishable from their use as register designators.

ASM processes each machine instruction line as follows: all labels are first equated to the location counter, '*'; then the opcode specified is inspected to determine which operand addressing modes are legal. Finally, the operand field is scanned for an appropriate addressing mode specification. The opcode is combined with the specified addressing mode to generate the object code corresponding to the desired instruction.

Many opcodes (LDA, STAA STX, SUBD, etc.) include a register specification (A, B, D, X, or S) as the last letter of the opcode mnemonic.

There are several operand modes for 68ØØ/68Ø1 instructions. A given instruction that is recognized by ASM will have one or more modes as legal forms (some opcodes require no operand specification whatsoever). The syntax of each of these modes is discussed in the following pages. A few examples of each mode will be given. A table of instruction mnemonics and their modes can be found at the end of this section.

Throughout this section, the notation " <expr> " means any expression, "EA=" means Effective Address, and "(" <expr> ")" to the right of "EA=" means "the contents of <expr>".

INHERENT MODE

Inherent Mode opcodes need no operand specifications. The argument field is ignored. For portability purposes, it should be left blank.

Syntax:

```
                <opcode>
```

Examples:

```
                RTI             ;Return from Interrupt
                DES             ;S:=S-1
                CLRA            ;A-Reg:=Ø
                PSHD            ;Pushes A and B
```

REGISTER-REGISTER MODE

Register-Register Mode addresses source and destination registers. This mode is used only for TFR A,B and TFR B,A 68Ø9 equivalent instructions. See M68ØC description.

Syntax:

```
                <opcode>  <reg>,<reg>
```

Examples:

```
                TFR     A,B
                Tfr     b,a
```

DIRECT MODE

Direct Mode is used to address a location in the range $ØØØØ through $ØØFF. The 8 bits of operand embedded in the instruction form the lower 8-bits of the 16-bit memory reference address while the upper 8-bits of the address are implicitly zero. If an instruction operand address evaluates within the range $ØØØØ to $ØØFF, ASM automatically generates Direct Mode memory reference if appropriate for that instruction.

Direct references may be forced with a "<" prefix. Use of this prefix prevents any default to the Extended Mode addressing. An error is generated if the effective address does not map into the range $ØØØØ to $ØØFF. Use of the "<" prefix is illegal with opcodes that only allow extended mode addressing.

Syntax:

```
                <opcode>        {<prefix>}<exp>
```

Example:

```
        A       EQU     $1Ø
        B       EQU     $123
                                ;Generated:
                SUBB    A       ;Direct reference to loc $1Ø
                INC     B       ;Extended reference to loc $123
                ADDB    <A      ;Direct reference to loc $1Ø
                LDA     <B      ;Direct ref to loc $123 WITH ERROR
                ROR     <A      ;Extended reference WITH ERROR
```

EXTENDED MODE

Extended mode addresses memory with a 16-bit address embedded in the instruction. Any location in the memory space can be referenced with this mode. Extended mode may be forced by use of the ">" prefix. Certain 6800/6801 instructions can only use Extended modes to directly reference memory. Instructions whose operand evaluates to an address in the range $0100 to $FFFF, or which are extended-mode only instructions are assembled with an extended address mode.

Syntax:

```
                <opcode>        {<prefix>}<exp>
```

Example:

```
        AA      EQU     $10
        BB      EQU     $1234
                LDAA    >AA     ;">" was required to produce
                                ;extended mode addressing
                STA     BB      ;Extended addressing
                ROR     AA      ;Always generates extended mode
```

EXTENDED INDIRECT MODE

Extended Indirect Mode addresses memory using a 16-bit address embedded in the instruction to retrieve the effective 16-bit address. This mode is indicated by "[" "]" surrounding the operand field.

Syntax:

```
                <opcode>        [<exp>]
```

Whenever indirect mode is encountered, ASM6800 substitutes the following:

```
                LDX             <exp>
                <opcode>        0,X
```

Note that use of indirect addressing destroys the contents of the X register. This is an extension of M6800 assembly code and is supported for M680C.

Example:

```
                LDD     [PNTR]  ;Load $1234 from loc QQ
                ...
        PNTR    FDB     QQ
                ...
        QQ      $1234
```

INDEXED MODE

Indexed mode addressing forms an effective address equal to the sum of an offset value and the contents of an index register.

There are several types of indexing that may be specified to ASM:

Zero offset
8-bit offset
Pre-decrement
Post-increment

All the above permit indirect addressing, specified by enclosing the operand field in "[" "]"s.

Indexed Mode always involves one of the following index register notations:

,X	Index register X
,S	System stack pointer

General syntax:

```
        <opcode>        {[}{{--}{<expr>},<indexreg>{++}{]}
```

Whenever indirect mode is encountered, ASM6800 substitutes the following:

```
                LDX             <expr>
                <opcode>        0,X
```

Note that use of indirect addressing destroys the contents of the X register. If the index register specified is S, then a TSX instruction is inserted, and the instruction treated as though ,X had been written instead.

ZERO OFFSET INDEXED MODE

This mode is also known as a register indirect addressing. The effective address is equal to the contents of the specified index register. This form generates a zero offset byte for a conventional 8 bit constant offset indexed mode.

Index Registers: X, S

Syntax:

```
                <opcode>        ,<reg>    ;(register indirect)
                <opcode>        [,<reg>]  ;(indirect register indirect)
```

Examples:

```
                LDA     ,X      ;EA = (X)
                LDB     [,S]    ;EA = ((S+1))
```

8-BIT CONSTANT OFFSET INDEXED MODE

Constant offset indexing forms an effective address equal to a constant plus the contents of an index register. The constant is embedded in the instruction. The constant must be in the range 0 to 255.

Index Registers: X, S

Syntax:

```
                <opcode>       <expr>,<reg>    ;constant offset
                <opcode>       [<expr>,<reg>]  ;indirect constant offset
```

Examples:

```
                SUBA     2,X              ;EA = (X)+5
                DEC      [61,S]           ;EA = ((S)+1+61)
                ADDD     [CAT,X]          ;EA = ((X)+CAT)
```

AUTO INCREMENT/DECREMENT INDEXED MODE

Index registers on the M68Ø9 may be automatically stepped by +1, +2, -1 and -2 bytes. ASM68ØØ simulates this for M68ØC. Increments are done AFTER the memory reference and hence annotated FOLLOWING the index register (i.e., X++); the effective address is the original contents of the index register. Decrements are done PRIOR to the memory reference and hence annotated PRECEEDING the index register (i.e., --Y); the effective address is the contents of the index register after it is decremented. Indirection is permitted, but only with the double stepped forms (++, --).

Index Registers: X, S

Syntax:

```
<opcode>        ,<reg>+
<opcode>        ,<reg>++
<opcode>        ,-<reg>
<opcode>        ,--<reg>
<opcode>        [,<reg>++]
<opcode>        [,--<reg>]
```

ASM68ØØ substitutes for each of the above:

```
<opcode>        ,<reg>  ; <reg>+
IN<reg>

<opcode>        ,<reg>  ; <reg>++
IN<reg>

IN<reg>
DE<reg>
<opcode>        ,<reg> ;-<reg>

DE<reg>                ;--<reg>
DE<reg>
<opcode>        ,<reg>
```

Indirect mode is treated as described previously.

Examples:

```
LDA     ,X+     ;EA=(X) \ X=X+1
STA     ,-S     ;Y=Y-1 \ EA=(Y)
LDD     ,X++    ;EA=(X) \ X=X+2
STX     ,-Y     ;Y=Y-2 \ EA=(Y)
LDX     [,S++]  ;EA=((Y)+2) \ Y=Y+2
```

RELATIVE MODE

The branch (Bxx and LBxx) class of instructions use this mode. There are two offset sizes used in relative mode, 8 and 16 bits. The 8 bit form is invoked with instruction mnemonics of the form "Bxx" and the 16 bit with "LBxx". The effective address is equal to the address of the next instruction plus the value of the (sign-extended) constant offset embedded in the instruction. The "LBxx" form generates 2 M6800 instructions: a conditional branch on the opposite condition around a JMP extended to the destination.

Syntax:

```
        Bxx     <expr>
        LBxx    <expr>
```

Examples:

```
                BRA     BLIMP
                LBCC    ZEPPELIN
```

NOTE: If the 8-bit form ("Bxx") is requested and the <expr> expression is evaluable on Pass 1 to a destination that is out of range, ASM will substitute the 16-bit ("LBxx") form.

IMMEDIATE MODE

Many 6800 instructions use a constant embedded in the instruction rather than an operand in a memory location separate from the instruction. This is designated "immediate" mode. The size of an immediate operand is determined by the instruction, not the operand; some instructions use 16 bit immediate operands while others use 8 bit immediate operands. The notation "#<expr>" is used to specify an immediate operand; if only 8 bits are required by the instruction, the expression value must be in the range -128 to 255 or an error will result.

Syntax:

```
                <opcode>        #<expr>
```

Examples:

```
                ADDA    #1              ; adds 1 to A register
                SUBD    #$4071
                LDS     #BUFFER+2

                SUBB    #BUFFER\256     ; same general effect as
                SBCA    #BUFFER/256     ; SUBD #BUFFER
```

OPCODE MNEMONICS RECOGNIZED BY ASM6800

This table lists all the opcode mnemonics recognized by ASM6800. The operand modes accepted by ASM for each of the opcodes are marked in the table. Additionally, there are notations, comments and opcode mnemonic classifications. The notations will show expansions if the mnemonic causes alternative code or multiple machine instructions to be generated. The opcode classes are as follows:

6800	- 6800 standard mnemonic.
6801	- 6801 standard mnemonic. 6800 instructions are substituted if WITH M6801 is not specified at assembly time.
6809	- 6809 only mnemonic. Generates equivalent 6801 instructions.
680C	- 680C mnemonic; supported in the 680C instruction set. One or more 6801 instructions may be substituted. Memory reference instructions are limited to a restricted subset of the 6809 indexed addressing forms. See Appendix D for more detail.

*OPERAND MODE KEY:

IDX=INDEXED	EXT=EXTENDED	DIR=DIRECT	IMM=IMMEDIATE
INH=INHERENT	BRA=BRANCH	PSH=PUSH/PULL	R/R=REG/REG

OPCODE	OPERAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	IDX	EXT	DIR	IMM	INH	BRA	PSH	R/R		
ABA	.	.	.	.	X	.	.	.	68ØØ	
ABX	.	.	.	.	X	.	.	.	68Ø1	See Footnote #1
ADCA	X	X	X	X	.	.	.	.	68ØØ	
ADCB	X	X	X	X	.	.	.	.	68ØØ	
ADCD	X	X	X	X	.	.	.	.	68ØC	ADCB arg+1 \ ADCA arg
ADDA	X	X	X	X	.	.	.	.	68ØØ	
ADDB	X	X	X	X	.	.	.	.	68ØØ	
ADDD	X	X	X	X	.	.	.	.	68Ø1	ADDB arg+1 \ ADCA arg
ANDA	X	X	X	X	.	.	.	.	68ØØ	
ANDB	X	X	X	X	.	.	.	.	68ØØ	
ANDD	X	X	X	X	.	.	.	.	68ØØ	ANDB arg+1 \ ANDA arg
ASL	X	X	.	.	.	.	.	.	68ØØ	
ASLA	.	.	.	.	X	.	.	.	68ØØ	
ASLB	.	.	.	.	X	.	.	.	68ØØ	
ASLD	.	.	.	.	X	.	.	.	68Ø1	ASLB \ ASLA
ASR	X	X	.	.	.	.	.	.	68ØØ	
ASRA	.	.	.	.	X	.	.	.	68ØØ	
ASRB	.	.	.	.	X	.	.	.	68ØØ	
ASRD	.	.	.	.	X	.	.	.	68ØC	ASRA \ RORB
BCC	.	.	.	.	.	X	.	.	68ØØ	
BCS	.	.	.	.	.	X	.	.	68ØØ	
BEQ	.	.	.	.	.	X	.	.	68ØØ	
BEQD	.	.	.	.	.	X	.	.	68ØC	BNE xxx \ TSTA \ xxx BEQ arg
BGE	.	.	.	.	.	X	.	.	68ØØ	
BGT	.	.	.	.	.	X	.	.	68ØØ	
BHI	.	.	.	.	.	X	.	.	68ØØ	
BHS	.	.	.	.	.	X	.	.	68ØØ	
BITA	X	X	X	X	.	.	.	.	68ØØ	
BITB	X	X	X	X	.	.	.	.	68ØØ	
BLE	.	.	.	.	.	X	.	.	68ØØ	
BLO	.	.	.	.	.	X	.	.	68ØØ	
BLS	.	.	.	.	.	X	.	.	68ØØ	
BLT	.	.	.	.	.	X	.	.	68ØØ	
BMI	.	.	.	.	.	X	.	.	68ØØ	
BNE	.	.	.	.	.	X	.	.	68ØØ	
BNED	.	.	.	.	.	X	.	.	68ØC	BEQ xxx \ TSTA \ xxx BNE arg
BPL	.	.	.	.	.	X	.	.	68ØØ	
BRA	.	.	.	.	.	X	.	.	68ØØ	
BRN	.	.	.	.	.	X	.	.	68Ø1	NOP \ NOP
BSR	.	.	.	.	.	X	.	.	68ØØ	
BVC	.	.	.	.	.	X	.	.	68ØØ	
BVS	.	.	.	.	.	X	.	.	68ØØ	

OPCODE	OPERAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	IDX	EXT	DIR	IMM	INH	BRA	PSH	R/R		
CBA	.	.	.	.	X	.	.	.	68ØØ	
CLC	.	.	.	.	X	.	.	.	68ØØ	
CLI	.	.	.	.	X	.	.	.	68ØØ	
CLR	X	X	.	.	.	.	.	.	68ØØ	
CLRA	.	.	.	.	X	.	.	.	68ØØ	
CLRB	.	.	.	.	X	.	.	.	68ØØ	
CLV	.	.	.	.	X	.	.	.	68ØØ	
CMPA	X	X	X	X	.	.	.	.	68ØØ	
CMPB	X	X	X	X	.	.	.	.	68ØØ	
CMPD	X	X	X	X	.	.	.	.	68ØC	See Footnote 2
CMPX	X	X	X	X	.	.	.	.	68Ø9	CPX
COM	X	X	.	.	.	.	.	.	68ØØ	
COMA	.	.	.	.	X	.	.	.	68ØØ	
COMB	.	.	.	.	X	.	.	.	68ØØ	
COMD	.	.	.	.	X	.	.	.	68ØC	COMB \ COMA
CPX	X	X	X	X	.	.	.	.	68ØØ	
DAA	.	.	.	.	X	.	.	.	68ØØ	
DEC	X	X	.	.	.	.	.	.	68ØØ	
DECA	.	.	.	.	X	.	.	.	68ØØ	
DECB	.	.	.	.	X	.	.	.	68ØØ	
DECD	X	X	X	.	.	.	.	.	68ØC	See Footnote 3
DES	.	.	.	.	X	.	.	.	68ØØ	
DEX	.	.	.	.	X	.	.	.	68ØØ	
EORA	X	X	X	X	.	.	.	.	68ØØ	
EORB	X	X	X	X	.	.	.	.	68ØØ	
EORD	X	X	X	X	.	.	.	.	68ØC	EORB arg+1 \ EORA arg
ERRORRTS	.	.	.	.	X	.	.	.	68ØC	SEC \ RTS
INC	X	X	.	.	.	.	.	.	68ØØ	
INCA	.	.	.	.	X	.	.	.	68ØØ	
INCB	.	.	.	.	X	.	.	.	68ØØ	
INCD	X	X	X	.	.	.	.	.	68ØC	See Footnote 4
INS	.	.	.	.	X	.	.	.	68ØØ	
INX	.	.	.	.	X	.	.	.	68ØØ	

OPCODE	OPERAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	I D X	E X T	D I R	I M M	I N H	B R A	P S H	R / R		
JMP	X	X	.	.	.	.	.	.	68ØØ	
JSR	X	X	X	.	.	.	.	.	68ØØ	
LBCC	.	.	.	.	.	X	.	.	68ØC	BCS xxx \ JMP arg \ xxx ...
LBCS	.	.	.	.	.	X	.	.	68ØC	BCC xxx \ JMP arg \ xxx ...
LBEQ	.	.	.	.	.	X	.	.	68ØC	BNE xxx \ JMP arg \ xxx ...
LBGE	.	.	.	.	.	X	.	.	68ØC	BLT xxx \ JMP arg \ xxx ...
LBGT	.	.	.	.	.	X	.	.	68ØC	BLG xxx \ JMP arg \ xxx ...
LBHI	.	.	.	.	.	X	.	.	68ØC	BLS xxx \ JMP arg \ xxx ...
LBHS	.	.	.	.	.	X	.	.	68ØC	BLO xxx \ JMP arg \ xxx ...
LBLE	.	.	.	.	.	X	.	.	68ØC	BGT xxx \ JMP arg \ xxx ...
LBLO	.	.	.	.	.	X	.	.	68ØC	BHS xxx \ JMP arg \ xxx ...
LBLS	.	.	.	.	.	X	.	.	68ØC	BHI xxx \ JMP arg \ xxx ...
LBLT	.	.	.	.	.	X	.	.	68ØC	BGE xxx \ JMP arg \ xxx ...
LBMI	.	.	.	.	.	X	.	.	68ØC	BPL xxx \ JMP arg \ xxx ...
LBNE	.	.	.	.	.	X	.	.	68ØC	BEQ xxx \ JMP arg \ xxx ...
LBRA	.	.	.	.	.	X	.	.	68ØC	JMP xxx
LBVC	.	.	.	.	.	X	.	.	68ØC	BVS xxx \ JMP arg \ xxx ...
LBVS	.	.	.	.	.	X	.	.	68ØC	BVS xxx \ JMP arg \ xxx ...
LDA	X	X	X	X	.	.	.	.	68ØØ	LDAA arg
LDAA	X	X	X	X	.	.	.	.	68ØØ	
LDAB	X	X	X	X	.	.	.	.	68ØØ	
LDB	X	X	X	X	.	.	.	.	68ØØ	LDAB arg
LDD	X	X	X	X	.	.	.	.	68Ø1	LDAB arg+1 \ LDAA arg
LDS	X	X	X	X	.	.	.	.	68ØØ	
LDX	X	X	X	X	.	.	.	.	68ØØ	
LEAS	X	.	.	.	.	.	.	.	68ØC	RPT arg \ INS
LEAX	X	.	.	.	.	.	.	.	68ØØ	RPT arg \ INX
LSL	X	X	.	.	.	.	.	.	68ØØ	ASL arg
LSLA	.	.	.	.	X	.	.	.	68ØØ	ASLA
LSLB	.	.	.	.	X	.	.	.	68ØØ	ASLB
LSLD	.	.	.	.	X	.	.	.	68ØC	ASLB \ ROLA
LSR	X	X	.	.	.	.	.	.	68ØØ	
LSRA	.	.	.	.	X	.	.	.	68ØØ	
LSRB	.	.	.	.	X	.	.	.	68ØØ	
LSRD	.	.	.	.	X	.	.	.	68ØC	LSRA \ RORB

OPCODE	OPERAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	IDX	EXT	DIR	IMM	INH	BRA	PSH	R/R		
MUL	.	.	.	.	X	.	.	.	68Ø1	JSR MUL68Ø9
NEG	X	X	.	.	.	.	.	.	68ØØ	
NEGA	.	.	.	.	X	.	.	.	68ØØ	
NEGB	.	.	.	.	X	.	.	.	68ØØ	
NEGD	.	.	.	.	X	.	.	.	68ØC	NEGA \ NEGB \ SBCA #Ø
NOP	.	.	.	.	X	.	.	.	68ØØ	
OKRTS	.	.	.	.	X	.	.	.	68ØC	CLC \ RTS
ORA	X	X	X	X	.	.	.	.	68ØØ	ORA
ORAA	X	X	X	X	.	.	.	.	68ØØ	
ORAB	X	X	X	X	.	.	.	.	68ØØ	
ORB	X	X	X	X	.	.	.	.	68ØØ	ORB
ORD	X	X	X	X	.	.	.	.	68ØC	ORB arg+1 \ ORA arg
PSHA	.	.	.	.	X	.	X	.	68ØØ	
PSHB	.	.	.	.	X	.	X	.	68ØØ	
PSHD	.	.	.	.	X	.	X	.	68ØC	PSHB \ PSHA
PSHX	.	.	.	.	X	.	X	.	68Ø1	See Footnote 5
PULA	.	.	.	.	X	.	X	.	68ØØ	
PULB	.	.	.	.	X	.	X	.	68ØØ	
PULD	.	.	.	.	X	.	X	.	68ØC	PULA \ PULB
PULX	.	.	.	.	X	.	X	.	68Ø1	TSX \ LDX Ø,X \ INS \ INS
ROL	X	X	.	.	.	.	.	.	68ØØ	
ROLA	.	.	.	.	X	.	.	.	68ØØ	
ROLB	.	.	.	.	X	.	.	.	68ØØ	
ROLD	.	.	.	.	X	.	.	.	68ØC	ROLB \ ROLA
ROR	X	X	.	.	.	.	.	.	68ØØ	
RORA	.	.	.	.	X	.	.	.	68ØØ	
RORB	.	.	.	.	X	.	.	.	68ØØ	
RORD	.	.	.	.	X	.	.	.	68ØC	RORA \ RORB
RTI	.	.	.	.	X	.	.	.	68ØØ	
RTS	.	.	.	.	X	.	.	.	68ØØ	
SBA	.	.	.	.	X	.	.	.	68ØØ	
SBCA	X	X	X	X	.	.	.	.	68ØØ	
SBCB	X	X	X	X	.	.	.	.	68ØØ	
SBCD	X	X	X	X	.	.	.	.	68ØC	SBCB arg+1 \ SBCA arg
SEC	.	.	.	.	X	.	.	.	68ØØ	

OPCODE	OPRAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	IDX	EXT	DIR	IMM	INH	BRA	PSH	R/R		
SEI	.	.	.	.	X	.	.	.	68ØØ	
SEV	.	.	.	.	X	.	.	.	68ØØ	
STA	X	X	X	.	.	.	.	.	68ØØ	STAA arg
STAA	X	X	X	.	.	.	.	.	68ØØ	
STAB	X	X	X	.	.	.	.	.	68ØØ	
STB	X	X	X	.	.	.	.	.	68ØØ	STAB arg
STD	X	X	X	.	.	.	.	.	68Ø1	STAB arg+1 \ STAA arg
STS	X	X	X	.	.	.	.	.	68ØØ	
STX	X	X	X	.	.	.	.	.	68ØØ	
SUBA	X	X	X	X	.	.	.	.	68ØØ	
SUBB	X	X	X	X	.	.	.	.	68ØØ	
SUBD	X	X	X	X	.	.	.	.	68Ø1	SUBB arg+1 \ SBCA arg
SWI	.	.	.	.	X	.	.	.	68ØØ	
TAB	.	.	.	.	X	.	.	.	68ØØ	
TAP	.	.	.	.	X	.	.	.	68ØØ	
TBA	.	.	.	.	X	.	.	.	68ØØ	
TDS	.	.	.	.	X	.	.	.	68ØC	STD TEMPX \ LDS TEMPX
TDX	.	.	.	.	X	.	.	.	68ØC	TFR D,X
TFR	.	.	.	.	.	.	.	X	68ØC	TAB or TBA
TPA	.	.	.	.	X	.	.	.	68ØØ	
TSD	.	.	.	.	X	.	.	.	68ØC	TSX \ TXD
TST	X	X	.	.	.	.	.	.	68ØØ	
TSTA	.	.	.	.	X	.	.	.	68ØØ	
TSTB	.	.	.	.	X	.	.	.	68ØØ	
TSTD	.	.	.	.	X	.	.	.	68ØC	SUBD #$ØØØØ
TSX	.	.	.	.	X	.	.	.	68ØØ	
TXD	.	.	.	.	X	.	.	.	68ØC	STX TEMPX \ LDD TEMPX
TXS	.	.	.	.	X	.	.	.	68ØØ	
WAI	.	.	.	.	X	.	.	.	68ØØ	

Footnote 1: STAB TEMPX+1\ADDB TEMPX+1\STAB TEMPX+1\
LDAB TEMPX \ ADCB #Ø\STAB TEMPX\LDX TEMPX

Footnote 2: CMPB arg+1 \ BNE xxx \ CMPA arg \ xxx ...

Footnote 3: TST arg+1 \ BNE xxx \ DEC arg \ xxx DEC arg+1

Footnote 4: INC arg+1 \ BNE xxx \ INC arg \ xxx ...

Footnote 5: STX TEMPX \ LDAB TEMP+1 \ PSHB \ LDAB TEMPX \ PSHB

68Ø9 MACHINE INSTRUCTION LINES

The major function of an assembler is the translation of symbolically specified machine instructions into a form directly understandable by the target computer. This section presumes knowledge of the 68Ø9 instruction set, which can be found in the Motorola MC68Ø9 Programming Manual. ASM is fully compatible with the Motorola 68Ø9 standard assembler syntax, with several useful extensions.

--All M68ØØ/M68Ø1 Opcodes are accepted and generate equivalent 68Ø9 code.

--Special set of opcodes to allow coding to work on 68ØØ, 68Ø1 and 68Ø9. These opcodes are called 68ØC opcodes (See Appendix D). Many "convenience" instructions are included in 68ØC, which act as logical extensions to the 'ØØ/'Ø9 instruction set, such as double register shifts, 16 bit memory increments, and decrements, etc.

--Automatic long branching. Short branches that are out-of-range will be assembled as long branches if the destination expression contains no forward references.

--Special offset prefix "<<" to force 5 bit offset form.

--Multiple labels are allowed, separated by commas. All labels present on an opcode line will be equated to the location counter, "*", before the rest of the line is processed.

--A, B, D, S, X, U, CC, DP, PC, and PCR may be used as ordinary symbols as long as they are distinguishable from their use as register designators.

ASM processes each machine instruction line as follows: all labels are first equated to the location counter, '*'; then the opcode specified is inspected to determine which operand addressing modes are legal. Finally, the operand field is scanned for an appropriate addressing mode specification. The opcode is combined with the specified addressing mode to generate the object code corresponding to the desired instruction.

Many opcodes (LDA, STX, CMPD, etc.) include a register specification (A, B, D, X, Y, U, or S) as the last letter of the opcode mnemonic.

There are several operand modes for 68Ø9 instructions. A given instruction will have one or more modes as legal forms (some opcodes require no operand specification whatsoever). The syntax of each of these modes is discussed in the following pages. A few examples of each mode will be given. A table of instruction mnemonics and their modes is given at the end of this section.

Throughout this section, the notation "<expr>" means any expression, "EA=" means Effective Address, and "(" ")" to the right of "EA=" means "the contents of".

INHERENT MODE

Inherent Mode opcodes need no operand specifications. The argument field is ignored. For portability purposes, it should be left blank.

Syntax:

```
        <opcode>
```

Examples:

```
        RTI                 ;Return from Interrupt
        MUL                 ;(A-Reg)*(B-Reg)
        CLRA                ;A-Reg:=Ø
```

REGISTER-REGISTER MODE

Register-Register Mode addresses source and destination registers. The registers come in two sizes, 8-bit and 16-bit. Only registers of like sizes may be addressed together. This mode may only be used with TFR or EXG opcodes.

16-bit Registers	8-bit Registers
X (X index reg)	A (A-reg)
Y (Y index reg)	B (B-reg)
U (User Stack Pointer)	CC (Condition Codes)
S (System Stack Pointer)	DP (Data Page Register)
PC (Program Counter)	
D (A+B register)	

Syntax:

```
        <opcode>  <reg>,<reg>
```

Examples:

```
        TFR     X,Y
        EXG     A,B
        Tfr     X,A      ;Illegal: X&A not same size
```

DIRECT MODE

Direct Mode is used to address a location in the 256 byte memory "page" designated by the contents of the DPR (Data Page Register). The 8 bits of operand embedded in the instruction form the lower 8-bits of the 16-bit memory reference address while the upper 8-bits of the address are supplied by the DPR. ASM must be informed what contents to assume in the DPR. This is done via the SETDPR directive (see Directives section). If an instruction operand address evaluates within the DPR page bounds, ASM automatically generates Direct Mode memory reference.

Direct references may be forced with a "<" prefix. Use of this prefix prevents any default to the Extended Mode addressing. If the effective address does not map into the Data Page as specified by the last SETDPR directive, an error is generated.

Syntax:

```
                <opcode>        {<prefix>}<exp>
```

Example:

```
        A       EQU     $10
        B       EQU     $123
        C       EQU     $456
                SETDPR  $100    ;PAGE 1 ($100-$1FF)
                                ;Generated:
                CLR     A       ;Extended reference to loc $10
                INC     B       ;Direct reference to loc $123
                DEC     C       ;Extended reference to $456
                ROR     <A      ;Direct reference to loc $110 WITH ERROR
                ASL     <B      ;Direct reference to loc $123
                TST     <C      ;Direct reference to loc $156 WITH ERROR
```

Note: The programmer is responsible for insuring that the value in the DPR during program execution matches the (SETDPR value)/$100.

EXTENDED MODE

Extended mode addresses memory with a 16-bit address embedded in the instruction. Any location in the memory space can be referenced with this mode. Extended mode may be forced by use of the ">" prefix.

Syntax:

```
                <opcode>        {<prefix>}<exp>
```

Example:

```
        AA      EQU     $10
        BB      EQU     $1234
                SETDPR  $00     ;(Default DPR setting)
                LDA     >AA     ;">" was required to produce
                                ;extended mode addressing
                STA     BB      ;Extended addressing
```

EXTENDED INDIRECT MODE

Extended Indirect Mode addresses memory using a 16-bit address embedded in the instruction to retrieve the effective 16-bit address. This mode is indicated by "[" "]" surrounding the operand field.

Syntax:

```
                <opcode>        [<exp>]
```

Example:

```
                LDD     [PNTR]  ;Load $1234 from loc QQ
                ...
        PNTR    FDB     QQ
                ...
        QQ      $1234
```

INDEXED MODE

Indexed mode addressing forms an effective address equal to the sum of an offset value and the contents of an index register.

There are many types of indexing that may be specified to ASM:

- Zero offset
- 5-bit offset
- 8-bit offset
- 16-bit offset
- A-Reg offset
- B-Reg offset
- D-Reg offset
- Pre-decrement
- Post-increment

All the above except 5-bit offset permit indirect addressing, specified by enclosing the operand field in "[" "]"s.

Indexed Mode always involves one of the following index register notations:

,X	Index register X
,Y	Index register Y
,S	System stack pointer
,U	User stack pointer
,PC	Program Counter
,PCR	Special form of ,PC for easy relative addressing (See section on PC relative addressing that follows)

General syntax:

```
        <opcode>          {<prefix>}{[}{{--}{<expr>},<indexreg>{++}{]}
```

ZERO OFFSET INDEXED MODE

This mode is also known as a register indirect addressing. The effective address is equal to the contents of the specified index register. Zero Offset Indexed Mode is the shortest and fastest M68Ø9 form for addressing via a register.

Index Registers: X, Y, U, S

Syntax:

```
                <opcode>       ,<reg>    ;(register indirect)
                <opcode>       [,<reg>]  ;(indirect register indirect)
```

Examples:

```
                LDA     ,X       ;EA = (X)
                LDB     [,Y]     ;EA = ((Y))
```

5-BIT, 8-BIT AND 16-BIT CONSTANT OFFSET INDEXED MODES

Constant offset indexing forms an effective address equal to a constant plus the contents of an index register. The constant is embedded in the instruction. The constant may be positive or negative.

There are several sizes of constant offsets available on the M6809. Notationally their invocations are identical. ASM attempts to assemble the shortest form. The limitations of the two-pass assembly technique force ASM to assume worst-case, 16-bit offset, for those offset expressions containing forward references. Prefix notations are provided to allow the programmer to force the offset size (5, 8 or 16 bits), even in the presence of a forward reference.

Index Registers: X, Y, U, S, PC

Syntax:

```
        <opcode>       <prefix><expr>,<reg>    ;constant offse
        <opcode>       [<prefix><expr>,<reg>]
                       ;indirect constant offset (Illegal for 
```

Examples:

```
        SUBA    2,X               ;EA = (X)+5
        DEC     [-61,Y]           ;EA = ((Y)-61)
        STY     27083,U           ;EA = (U)+27083
        LEAX    DOG,S             ;EA = (S)+DOG
        ADDD    [CAT,Y]           ;EA = ((Y)+CAT)
        LDD     FRED-*',PC        ;EA = FRED = PC+offset to FRED
        STD     [JOE-*;PC]        ;EA = (JOE) = (PC+offset to JOE
```

Special operand prefix notations:

```
        <<      force 5-bit offset (illegal with indirection)
        <       force 8-bit offset
        >       force 16-bit offset
```

Examples:

```
        ORB     <<BUGOUT,U        ;Assumes BUGOUT fits in 5 bits
        SUBB    >[FAROUT,X]       ;Forces offset to 16 bits
        LDA     <WAYOUT,Y         ;Assumes WAYOUT fits in 8 bits
```

Notes:

1. PC indexing does not have a 5-bit offset form. This means <<...,PC and <<...,PCR are illegal.

2. There is no 5-bit offset indirect form. This means <<[...] is illegal.

3. If the offset expression has no forward references and evaluates to zero during Pass 1, then the zero offset form will be substituted. The zero offset form saves one machine cycle over the 5-bit offset form.

   Example:

```
BACKWARD        EQU     0
        LDA     BACKWARD,X      ;Generates "LDA ,X"
        STA     FORWARD,Y       ;Generates "STA >0,Y"
FORWARD EQU     0
```

   Notice that the FORWARD reference generated a 16-bit offset. This is because in ASM Pass 1, the value of FORWARD was not known and the worst case was assumed.

PROGRAM COUNTER RELATIVE (INDEXED) MODE

This is a special form of constant offset indexing from the program counter. It is an alternate to the form "...,PC". The section "IMPLICIT VALUES" describes the *' value as being the location of the NEXT instruction. This is the implicit value of the PC during the execution of any instruction. If only the ",PC" form were available, PC relative addressing would usually be "<destination>-*',PC". Program Counter Relative notation produces the same result from "<destination>,PCR". Indirection is permitted in this mode. The prefixes "<" and ">" are valid.

Syntax:

```
        <opcode>        <expr>,PCR
        <opcode>        [<expr>,PCR]
```

Examples:

```
        ORA     MASK,PCR        ;EA=MASK
        ADDD    VALX,PCR        ;EA=VALX
        JMP     <THERE,PCR      ;Force 8-bit offset
        JSR     >[BEEP,PCR]     ;Force 16-bits and indirection
```

ACCUMULATOR OFFSET INDEXED MODE

Offsets on the M6809 may be specified to be the contents of an accumulator. The effective address is formed by 2's complement addition of the accumulator contents, sign extended to 16 bits, and the index register contents. Indirection may be applied to this indexing mode.

Index Registers: X, Y, U, S

Offset Accumulators: A, B, D

Syntax:

```
<opcode>        <acc>,<reg>
<opcode>        [<acc>,<reg>]
```

Examples:

```
LDA     B,X     ;EA = (B)+(X)
CLR     [A,Y]   ;EA = ((A)+(Y))
LDD     D,U     ;EA = (D)+(U)
```

AUTO INCREMENT/DECREMENT INDEXED MODE

Index registers on the M6809 may be automatically stepped by +1, +2, -1 and -2 bytes. Increments are done AFTER the memory reference and hence annotated FOLLOWING the index register (i.e., X++); the effective address is the original contents of the index register. Decrements are done PRIOR to the memory reference and hence annotated PRECEEDING the index register (i.e., --Y); the effective address is the contents of the index register after it is decremented. Indirection is permitted, but only with the double stepped forms (++, --).

Syntax:

```
<opcode>        ,<reg>+
<opcode>        ,<reg>++
<opcode>        ,-<reg>
<opcode>        ,--<reg>
<opcode>        [,<reg>++]
<opcode>        [,--<reg>]
```

Examples:

```
LDA     ,X+     ;EA=(X) \ X=X+1
STA     ,-Y     ;Y=Y-1 \ EA=(Y)
LDD     ,X++    ;EA=(X) \ X=X+2
STX     ,-Y     ;Y=Y-2 \ EA=(Y)
LDX     [,Y++]  ;EA=((Y)+2) \ Y=Y+2
```

RELATIVE MODE

The branch (Bxx and LBxx) class of instructions use this mode. There are two offset sizes used in relative mode, 8 and 16 bits. The 8 bit form is invoked with instruction mnemonics of the form "Bxx" and the 16 bit with "LBxx". The effective address is equal to the address of the next instruction plus the value of the (sign-extended) constant offset embedded in the instruction.

Syntax:

```
        Bxx     <expr>
        LBxx    <expr>
```

Examples:

```
                BRA     BLIMP
                LBCC    ZEPPELIN
```

NOTE: If the 8-bit form ("Bxx") is requested and the <expr> expression is evaluable on Pass 1 to a destination that is out of range, ASM will substitute the 16-bit ("LBxx") form.

IMMEDIATE MODE

Many 68Ø9 instructions use a constant embedded in the instruction rather than an operand in a memory location separate from the instruction. This is designated "immediate" mode. The size of an immediate operand is determined by the instruction, not the operand; some instructions use 16 bit immediate operands while others use 8 bit immediate operands. The notation "#<expr>" is used to specify an immediate operand; if only 8 bits are required by the instruction, the expression value must be in the range -128 to 255 or an error will result.

Syntax:

```
                <opcode>        #<expr>
```

Examples:

```
                ADDA    #1              ; adds 1 to A register
                CMPD    #$4Ø71
                LDY     #BUFFER+2

                SUBB    #BUFFER\256     ; same general effect
                SBCA    #BUFFER/256     ; as SUBD #BUFFER
```

STACK MODE

This mode may only be used with PSHS, PSHU, PULS, and PULU instructions. The operands in this mode are registers to be pushed or pulled from a stack (user or system). The operand field consists of a sequence of register names separated by commas or a single immediate mode expression. The ordering of registers is arbitrary since the order of PUSH/PULL is fixed. Mention of a register name sets the appropriate corresponding bit in the postbyte of the instruction. If an immediate expression is used, the lower 8 bits of the expression are used as the post byte. The immediate form has the advantage of allowing register groups to be symbolically named.

Registers: PC, S, U, Y, X, DP, B, A, D, CC

Syntax:

```
                <opcode>        <reg>,...,<reg>
                <opcode>        #<expr>
```

Examples:

```
                PSHU    PC,S,D,DP
                PULS    #STKFRAME       STKFRAME is some register subset
```

NOTE: The use of the "D" register is equivalent to "A,B".

OPCODE MNEMONICS RECOGNIZED BY ASM68Ø9

This table lists all the opcode mnemonics recognized by the ASM68Ø9. The operand modes accepted by ASM for each of the opcodes are marked in the table. Additionally, there are notations, comments and opcode mnemonic classifications. The notations will show expansions if the mnemonic causes an alternative code or multiple machine instructions to be generated. The opcode classes are as follows:

68Ø9	- 68Ø9 (Motorola) standard mnemonic. Generates a conventional 68Ø9 instruction.
68ØØ	- 68ØØ standard mnemonic which has an exact 68Ø9 counterpart. The 68Ø9 counterpart is generated.
68ØØEQ	- 68ØØ equivalent mnemonic; there is no exact 68Ø9 counterpart. Notes and comments will show the 68Ø9 instruction sequence substituted. The sequence is as close to functionally equivalent to the 68ØØ mnemonic as possible.
68ØC	- 68ØC mnemonic; supported in the 68ØC instruction set. One or more 68Ø9 instructions may be substituted. Memory reference instructions are limited to a restricted subset of the 68Ø9 indexed addressing forms. See Appendix D for more detail.

*OPERAND MODE KEY:

IDX=INDEXED	EXT=EXTENDED	DIR=DIRECT	IMM=IMMEDIATE
INH=INHERENT	BRA=BRANCH	PSH=PUSH/PULL	R/R=REG/REG

OPCODE	OPERAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	I	E	D	I	I	B	P	R		
	D	X	I	M	N	R	S	/		
	X	T	R	M	H	A	H	R		
ABA	.	.	.	.	X	.	.	.	68ØØEQ/68ØC	PSHS B\ ADDA ,S+
ABX	.	.	.	.	X	.	.	.	68Ø9/68ØC	
ADCA	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
ADCB	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
ADCD	X	X	X	X	.	.	.	.	68ØC	ADCB arg+1 \ADCA arg
ADDA	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
ADDB	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
ADDD	X	X	X	X	.	.	.	.	68Ø9/68ØC	
ANDA	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
ANDB	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
ANDC	.	.	.	X	.	.	.	.	68Ø9	Alternative for ANDCC
ANDCC	.	.	.	X	.	.	.	.	68Ø9	
ANDD	X	X	X	X	.	.	.	.	68ØC	ANDB arg+1 \ANDA arg
ASL	X	X	X	.	.	.	.	.	68Ø9/68ØØ	
ASLA	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
ASLB	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
ASLD	.	.	.	.	X	.	.	.	68ØC	ASLB \ ASLA
ASR	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
ASRA	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
ASRB	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
ASRD	.	.	.	.	X	.	.	.	68ØC	ASRA \ RORB
BCC	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BCS	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BEQ	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BEQD	.	.	.	.	.	X	.	.	68ØC	
BGE	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BHI	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BHS	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BITA	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
BITB	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
BLE	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BLO	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BLS	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BLT	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BMI	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BNE	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BNED	.	.	.	.	.	X	.	.	68ØC	
BPL	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BRA	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BRN	.	.	.	.	.	X	.	.	68Ø9	
BSR	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BVC	.	.	.	.	.	X	.	.	68Ø9/68ØØ	
BVS	.	.	.	.	.	X	.	.	68Ø9/68ØØ	

OPCODE	OPERAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	IDX	EXT	DIR	IMM	INH	BRA	PSH	R/R		
CBA	.	.	.	.	X	.	.	.	6800EQ/680C	PSHS B\CMPA ,S+
CLC	.	.	.	.	X	.	.	.	6800EQ	ANDCC #$FE
CLI	.	.	.	.	X	.	.	.	6800EQ	ANDCC #$EF
CLR	X	X	X	.	.	.	.	.	6809/6800	
CLRA	.	.	.	.	X	.	.	.	6809/6800	
CLRB	.	.	.	.	X	.	.	.	6809/6800	
CLV	.	.	.	.	X	.	.	.	6800EQ	ANDCC #$FD
CMPA	X	X	X	X	.	.	.	.	6809/6800	
CMPB	X	X	X	X	.	.	.	.	6809/6800	
CMPD	X	X	X	X	.	.	.	.	6809/680C	
CMPS	X	X	X	X	.	.	.	.	6809	
CMPU	X	X	X	X	.	.	.	.	6809	
CMPX	X	X	X	X	.	.	.	.	6809/6800	
CMPY	X	X	X	X	.	.	.	.	6809	
COM	X	X	X	.	.	.	.	.	6809/6800	
COMA	.	.	.	.	X	.	.	.	6809/6800	
COMB	.	.	.	.	X	.	.	.	6809/6800	
COMD	.	.	.	.	X	.	.	.	680C	COMB \ COMA
CPX	X	X	X	X	.	.	.	.	680EQ	CMPX
CWAI	.	.	.	X	.	.	.	.	6809	
DAA	.	.	.	.	X	.	.	.	6809/6800	
DEC	X	X	X	.	.	.	.	.	6809/6800	
DECA	.	.	.	.	X	.	.	.	6809/6800	
DECB	.	.	.	.	X	.	.	.	6809/6800	
DECD	X	X	X	.	.	.	.	.	680C	See Footnote 1
DES	.	.	.	.	X	.	.	.	680EQ	LEAS -1,S
DEX	.	.	.	.	X	.	.	.	680EQ	LEAX -1,X
EORA	X	X	X	X	.	.	.	.	6809/6800	
EORB	X	X	X	X	.	.	.	.	6809/6800	
EORD	X	X	X	X	.	.	.	.	680C	EORB arg+1 \ EORA arg
ERRORRTS	.	.	.	.	X	.	.	.	680C	ORCC #1 \ RTS
EXG	.	.	.	.	.	.	.	X	6809	
INC	X	X	X	.	.	.	.	.	6809/6800	
INCA	.	.	.	.	X	.	.	.	6809/6800	
INCB	.	.	.	.	X	.	.	.	6809/6800	
INCD	X	X	X	.	.	.	.	.	680C	See Footnote 2
INS	.	.	.	.	X	.	.	.	680EQ	LEAS 1,S
INX	.	.	.	.	X	.	.	.	680EQ	LEAX 1,X

OPCODE	OPERAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	IDX	EXT	DIR	IMM	INH	BRA	PSH	R/R		
JMP	X	X	X	.	.	.	.	.	68Ø9/68ØØ/68ØC	
JSR	X	X	X	.	.	.	.	.	68Ø9/68ØØ/68ØC	
LBCC	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBCS	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBEQ	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBGE	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBGT	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBHI	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBHS	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBLE	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBLO	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBLS	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBLT	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBMI	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBNE	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBRA	.	.	.	.	.	X	.	.	68Ø9	
LBRN	.	.	.	.	.	X	.	.	68Ø9	
LBSR	.	.	.	.	.	X	.	.	68Ø9	
LBVC	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LBVS	.	.	.	.	.	X	.	.	68Ø9/68ØC	
LDA	X	X	X	X	.	.	.	.	68Ø9/68ØC	
LDAA	X	X	X	X	.	.	.	.	68ØØEQ	LDA
LDAB	X	X	X	X	.	.	.	.	68ØØEQ	LDB
LDB	X	X	X	X	.	.	.	.	68Ø9/68ØC	
LDD	X	X	X	X	.	.	.	.	68Ø9/68ØC	
LDS	X	X	X	X	.	.	.	.	68Ø9/68ØC	
LDU	X	X	X	X	.	.	.	.	68Ø9	
LDX	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
LDY	X	X	X	X	.	.	.	.	68Ø9	
LEAS	X	.	.	.	.	.	.	.	68Ø9/68ØC	
LEAU	X	.	.	.	.	.	.	.	68Ø9	
LEAX	X	.	.	.	.	.	.	.	68Ø9/68ØC	
LEAY	X	.	.	.	.	.	.	.	68Ø9	
LSL	X	X	X	.	.	.	.	.	68Ø9/68ØØ	
LSLA	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
LSLB	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
LSLD	.	.	.	.	X	.	.	.	68ØC	ASLB \ ROLA
LSR	X	X	X	.	.	.	.	.	68Ø9/68ØØ	
LSRA	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
LSRB	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
LSRD	.	.	.	.	X	.	.	.	68Ø9/68ØC	LSRA \ RORB

OPCODE	OPERAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	IDX	EXT	DIR	IMM	INH	BRA	PSH	R/R		
MUL	.	.	.	.	X	.	.	.	68Ø9/68ØC	
NEG	X	X	X	.	.	.	.	.	68Ø9/68ØØ	
NEGA	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
NEGB	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
NEGD	.	.	.	.	X	.	.	.	68ØC	NEGA \ NEGB \ SBCA #Ø
NOP	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
OKRTS	.	.	.	.	X	.	.	.	68ØC	ANDCC #$FE \ RTS
ORA	X	X	X	X	.	.	.	.	68Ø9/68ØC	
ORAA	X	X	X	X	.	.	.	.	68ØØEQ	ORA
ORAB	X	X	X	X	.	.	.	.	68ØØEQ	ORB
ORB	X	X	X	X	.	.	.	.	68Ø9/68ØC	
ORCC	.	.	.	X	.	.	.	.	68Ø9	
ORD	X	X	X	X	.	.	.	.	68ØC	ORB arg+1 \ ORA arg
PSHA	.	.	.	.	X	.	.	.	68ØØEQ	PSHS A
PSHB	.	.	.	.	X	.	.	.	68ØØEQ	PSHS B
PSHD	.	.	.	.	X	.	.	.	68ØC	PSHS D
PSHS	.	.	.	.	X	.	X	.	68Ø9	
PSHU	.	.	.	.	X	.	X	.	68Ø9	
PSHX	.	.	.	.	X	.	.	.	68ØC	PSHS X
PULA	.	.	.	.	X	.	.	.	68ØØEQ	PULS A
PULB	.	.	.	.	X	.	.	.	68ØØEQ	PULS B
PULD	.	.	.	.	X	.	.	.	68ØØEQ	PULS D
PULS	.	.	.	X	.	.	X	.	68Ø9	
PULU	.	.	.	X	.	.	X	.	68Ø9	
PULX	.	.	.	X	.	.	.	.	68ØC	PULS X
ROL	X	X	X	.	.	.	.	.	68Ø9/68ØØ	
ROLA	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
ROLB	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
ROLD	.	.	.	.	X	.	.	.	68ØC	ROLB \ ROLA
ROR	X	X	X	.	.	.	.	.	68Ø9/68ØØ	
RORA	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
RORB	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
RORD	.	.	.	.	X	.	.	.	68ØC	RORA \ RORB
RTI	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
RTS	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
SBA	.	.	.	.	X	.	.	.	68ØØEQ/68ØC	PSHS B \ SUBA ,S+
SBCA	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
SBCB	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
SBCD	X	X	X	X	.	.	.	.	68ØC	SBCB arg+1 \ SBCA arg
SEC	.	.	.	.	X	.	.	.	68ØØEQ	ORCC #$Ø1

OPCODE	OPRAND MODE*								OPCODE CLASS	NOTES AND COMMENTS
	IDX	EXT	DIR	IMM	INH	BRA	PSH	R/R		
SEI	.	.	.	.	X	.	.	.	68ØØEQ	ORCC #$1Ø
SEV	.	.	.	.	X	.	.	.	68ØØEQ	ORCC #$Ø2
SEX	.	.	.	.	X	.	.	.	68Ø9	
STA	X	X	X	.	.	.	.	.	68Ø9/68ØC	
STAA	X	X	X	.	.	.	.	.	68ØØEQ	STA
STAB	X	X	X	.	.	.	.	.	68ØØEQ	STB
STB	X	X	X	.	.	.	.	.	68Ø9/68ØC	
STD	X	X	X	.	.	.	.	.	68Ø9/68ØC	
STS	X	X	X	.	.	.	.	.	68Ø9/68ØØ	
STU	X	X	X	.	.	.	.	.	68Ø9	
STX	X	X	X	.	.	.	.	.	68Ø9/68ØØ	
STY	X	X	X	.	.	.	.	.	68Ø9	
SUBA	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
SUBB	X	X	X	X	.	.	.	.	68Ø9/68ØØ	
SUBD	X	X	X	X	.	.	.	.	68Ø9/68ØC	
SWI	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
SWI2	.	.	.	.	X	.	.	.	68Ø9	
SWI3	.	.	.	.	X	.	.	.	68Ø9	
SYNC	.	.	.	.	X	.	.	.	68Ø9	
TAB	.	.	.	.	X	.	.	.	68ØØEQ	TFR A,B \ TSTA
TAP	.	.	.	.	X	.	.	.	68ØØEQ	TFR A,CC
TBA	.	.	.	.	X	.	.	.	68ØØEQ	TFR B,A \ TSTA
TDS	.	.	.	.	X	.	.	.	68ØC	TFR D,S
TDX	.	.	.	.	X	.	.	.	68ØC	TFR D,X
TFR	.	.	.	.	.	.	.	X	68Ø9	
TPA	.	.	.	.	X	.	.	.	68ØØEQ	TFR CC,A
TSD	.	.	.	.	X	.	.	.	68ØC	TFR S,D
TST	X	X	X	.	.	.	.	.	68Ø9/68ØØ	
TSTA	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
TSTB	.	.	.	.	X	.	.	.	68Ø9/68ØØ	
TSTD	.	.	.	.	X	.	.	.	68ØC	SUBD #$ØØØØ
TSX	.	.	.	.	X	.	.	.	68ØØEQ	TFR S,X
TXD	.	.	.	.	X	.	.	.	68ØC	TFR X,D
TXS	.	.	.	.	X	.	.	.	68ØØEQ	TFR X,S
WAI	.	.	.	.	X	.	.	.	68ØØEQ	CWAI #$FF

Footnote 1: TST arg+1\BNE xxx\DEC arg\xxx: DEC arg+1

Footnote 2: INC arg+1\BNE xxx\INC arg\xxx:

DIRECTIVES

Directives are used to control the action of ASM in ways not directly related to the generation of machine language opcodes. Throughout this section we will use the following notation to describe the syntax of directives.

<LF> indicates a list of zero or more symbols separated by commas, occuring in the label field of the line. The list may begin with a comma, and consecutive commas are allowed. The exact syntax is:

```
              { <SYMBOL> }  { , { <SYMBOL> } ... }
```

If such a list of symbols is malformed, "Illegal Label" will be reported.

<EXPLIST> indicates a list of expressions separated by commas. Consecutive commas, leading commas and trailing commas are allowed, and are interpreted as having a zero expression where an expression is missing. The exact definition of <EXPLIST> is:

```
              <EXP>   { , { <EXP> } ... }
                      , { <EXP> }
```

SYMBOL DEFINITION DIRECTIVES

Two commands are provided to allow users directly to assign a value to a symbol.

EQUATE

```
        {<LF>}  EQU     <EXP>
```

EQU directs ASM permanently to associate the value given as the argument with the symbols given in the label field. <EXP> may involve one level of forward reference; if it does, then the symbols specified in the label field will be treated as forward references throughout the assembly.

--Any attempt to redefine the value of an EQU'd symbol will be diagnosed as an error.

--If no symbols are present in the label field, EQU merely evaluates the expression. This can be useful to display the value of a given expression in the listing.

--If no expression is present, "Illegal Argument" is reported.

IMPLICIT EQU:

```
        <LF>
```

A label found in the label field, with a null command field, is treated as shorthand for

```
        <LF>    EQU     *
```

This allows a notationally pretty method of placing very long labels in sequences of assembled machine instructions.

Examples:

```
A,B     EQU     1Ø      ;Sets values of A and B to 1Ø.
        EQU     A       ;Displays value of A
C       EQU             ;Causes "illegal argument" error
D       EQU     E       ;One level of forward
E       EQU     $15     ;reference is allowed.
        LDAA    E       ;This will generate a page-zero
                        ;reference, 2-bytes
        LDAA    D       ;This will generate a long
                        ;reference, 3 bytes.

I       EQU     1Ø      ;Note that once a symbol is EQU'd,
I       EQU     99      ;it may not be redefined: this an erro

F       EQU     G       ;This is illegal...
G       EQU     H
H       EQU     $1Ø
        FCB     F       ; It is diagnosed here.

THISISAVERYLONGLABEL    ; Whose value is equated to *
```

SET

```
        {<LF>}  SET     <EXP>
```

SET is like EQU (see above) with one difference: symbols whose values have been defined by SET may later be redefined using SET. <EXP> may NOT involve a forward reference.

SET and EQU are mutually exclusive. If a symbol is SET in its first definition, it may not later be EQU'd; if a symbol is EQU'd first, it may not later be SET. Any violation of these rules will result in "Double Definition" being reported.

--Like EQU, SET does not require that labels be present. If no symbols are present in the label field, SET is equivalent to EQU. Note that forward references cannot be displayed with SET.

--If <EXP> involves a forward reference, "Illegal Value" is reported, and the value zero is used instead.

--If no argument field is present, "Illegal Argument" is reported.

--If an attempt to SET a symbol conflicts with another definition somewhere else in the assembly, "Double Definition" is reported. The first definition of the symbol is retained.

Examples:

```
A       SET     1Ø      ;Value of A is 1Ø
        ...
        FDB     A       ;Generates $ØØØA
        ...
A       SET     2Ø      ;Value of A is now 2Ø
        ...
        FDB     A       ;Generates $ØØ14

B       EQU     $5
        ...
B       SET     $1Ø     ;The first definition applies.

                        ;"Double def" is reported both
                        ;here and above.

;  Double-def does not prevent a symbol that was
;  SET first from being SET later.
C       SET     $7
C       EQU     $9      ;Double def!
        FCB     C       ;Generates $Ø7
C       SET     $A
        FCB     C       ;Generates $ØA

D       SET     E       ;This is illegal --
E       EQU     $15     ;D ends up set to zero.
```

DATA GENERATION DIRECTIVES

Several methods of generating data are provided by ASM.

FORM CONSTANT BYTE

```
        {<LF>}  FCB     { <EXPLIST> }
```

FCB directs ASM to output a series of one-byte values to be loaded into memory starting at the current value of the location counter. Expressions may be of any type; however, their values must be between -$8Ø and $FF.

--If labels are present, they are EQU'd to the location counter before data generation begins.

--Null expressions (i.e., contiguous commas) cause a zero value to be generated for that expression. No error is reported.

--Each expression is output as soon as it has been evaluated. This means that "*" will have a different value in each expression across the line.

Examples:

```
                FCB     1Ø,12    ;Generates $ØA followed by $ØC.
        A,B     FCB              ;A and B are EQU'd to *
                                 ;one byte of $ØØ is output.
                FCB     -3,7,'9 ;Note that leading sign is OK.
                FCB     ,,,3     ;Generates Ø,Ø,Ø,3
                FCB     $1Ø5     ;Reports overflow, generates $Ø5

                FCB     3,       ;Generates byte of 3, then byte of Ø
```

RESERVE MEMORY BYTES

```
        {<LF>}  RMB     <EXP>
```

RMB directs ASM to reserve memory space without initializing its contents. <EXP> is interpreted as a sixteen bit, positive integer. It may not be a forward reference.

--Any labels present will be EQU'd to the location counter before the RMB is processed.

--If <EXP> is a forward reference, "Illegal Value" is reported, and the location counter is not moved.

--If <EXP> is not present, "Illegal Argument" is reported, and the location counter is not moved.

--The location counter's old value and the number of bytes reserved are listed.

Examples:

```
ØØØØ ØØ3Ø              RMB     $3Ø     ;Reserve $3Ø bytes
  ØØ3Ø                 EQU     *

ØØ3Ø ØØ1Ø       A      RMB     $1Ø     ; A label
  ØØ4Ø                 EQU     *       ; is EQU'd to the
  ØØ3Ø                 EQU     A       ; first byte of the reserved space.

ØØ4Ø ØØØØ       B      RMB     Ø       ; Zero bytes can be reserved.
  ØØ4Ø                 EQU     B

ØØ4Ø ØØØØ              RMB     C       ;This is illegal!!
*** Illegal Value.
  ØØ4Ø          C      EQU     *       ;C was a forward reference.
```

FORM DOUBLE BYTE CONSTANT

```
{<LF>}  FDB     { <EXPLIST> }
```

FDB directs ASM to assemble the expressions given in the argument field into memory as two-byte values. The expressions may be of any type.

--Each expression is output as soon as it has been evaluated. This means that "*" will have a different value in each expression across a given line.

--If there are labels on the line, they will be EQU'd to the location counter before the line is processed.

--The most significant byte of each expression is output, followed by the least significant byte.

--Null expressions are treated as zero values as for FCB; two bytes of zero will be output.

--If more than 4 bytes are generated, only the first four are listed on the line with the statement. Additional lines are used to list all bytes after the fourth, and will be listed only if PGEN 1 has been specified.

Examples:

```
ØØØØ ØØØØ       A,B     FDB     Ø       ;A, B are EQU'd to *
  ØØØØ                  EQU     A

ØØØ2 ØØØ2ØØØ4   C       FDB     *,*     ;Note that *'s value changes

ØØØ6 ØØØØ               FDB             ;Generates 2 bytes of zero

ØØØ8 ØØØØØØØØ           FDB     ,,,3    ;Generates 6 zero bytes, then $ØØØ3
ØØØC ØØØØØØØ3

ØØ1Ø Ø427ØØØØ           FDB     $427,   ;Note trailing zero
```

FORM CONSTANT CHARACTERS

```
        {<LF>}  FCC     <DELIM> { <CHARACTER> ...} <DELIM>
```

FCC causes ASM to assemble the ASCII value of a string of characters into memory.

The first character of the argument string is taken as the delimiter, and may be any character (except carriage-return). All characters between the delimiter and the second occurence of the delimiter character are assembled as the string; the most significant bit of each character will be zero.

--If the line ends before a second occurence of the delimiter is found, "Illegal String" is reported, and the carriage return is treated as the end of the string.

--A digit may be used as the delimiter of a string, provided that the string does not begin with a series of digits followed by a comma.

--The string may contain any ASCII character with the exception of null and carriage return. This is an extension over the Motorola standard, which only allows characters whose values are between $2Ø and $5F.

--Tabs contained in the string will be printed as "^I".

--Any labels present will be EQU'd to the location counter before the line is processed.

--The first four bytes generated by the FCC command will be listed on the line listing the source statement. Further bytes will be listed as for FCB and FDB. Note that if PGEN 1 has not been specified, only the first four bytes will be listed.

To enhance portability to future SD assemblers, we strongly recommend use of " (double-quote) as the <DELIM> character.

Examples:

```
  ØØØ1                  PCC     1       ;So we can see the PGEN
  ØØØ1                  PGEN    1       ;List everything, please.

ØØØØ 53545249           FCC     "STRING"
ØØØ4 4E47

ØØØA 4E472Ø2Ø
  ØØØØ                  PGEN    Ø       ;List only first 4 bytes.
ØØØ6 4F4E4C59   K       FCC     'Only 4 bytes listed'
  ØØ13                  EQU     *-K     But all here generated.

ØØ19 Ø9                 FCC     /^I/    Look how nicely tabs get listed.
```

FORM FLOATING POINT NUMBER

```
        {<LF>}  FFC     <floatingpointnumber>
```

FFC (Form Floating Constant) causes ASM to convert the floating point number specified to its corresponding binary form and place the results into memory. Each floating point number is stored in the BASE 100 format used by the SD BASIC 1.4 Compiler; range is limited to 10**126. The syntax of numbers accepted is identical to forms accepted by the compiler. For more details, refer to the BASIC 1.4 manual.

--Any labels present will be EQU'd with the location counter before the line is processed.

--Unreasonably large numbers will get an overflow error and ASM's version of infinity will be substituted.

Examples:

```
0000 00000000     4:          FFC     0
0004 0000
0006 41010000     5:          FFC     1
000A 0000
000C 41030E0F     6: PI       FFC     3.14159287
0010 5C57
0012 C1010000     7:          FFC     -1
0016 0000
0018 530A0000     8:          FFC     1E37
001C 0000
001E 32624C36     9:          FFC     .000987654321E-25
0022 200A
```

IMPLICIT DATA STATEMENT

```
        {<LF>}  <EXPLIST>
```

ASM allows the user to generate data without writing FDB or FCB statements. If the command field of a line cannot be interpreted as a machine language instruction or a directive, ASM will interpret the entire command field as a list of expressions whose values are to be assembled as data.

Each expression will be assembled into one or two bytes, depending on its value. If an expression has a value between Ø and $FF, it will be assembled into one byte. Otherwise it will be assembled into two bytes.

These defaults can be overriden in the following ways:

1) If the expression is preceded by a hash mark ("#"), the hash mark will be ignored, and the expression will be assembled into two bytes regardless of its value.

2) If the expression is preceded by monadic plus or minus, the expression will be assembled into one byte regardless of value. The minus sign will have its usual effect.

--Labels present on the line will be EQU'd to the location counter before the statement is processed.

--If an expression contains a forward reference or an undefined symbol, it will be assembled into two bytes unless a prefix of "+" or "-" is present.

--Each expression is output as soon as it has been evaluated. This means that "*" will have a different value for each expression across the line.

Examples:

```
0000 07                     7                    ;Generates one byte.
0001 0105                   $105                 ;Generates two bytes.

  0011        EQU           EQU       17         Define symbol called EQU,
0003 11                     +EQU                 Generate the value like this,
0004 11                     (EQU)                Like this,
0005 0011                   #EQU                 or like this,
  0000                      EQU                  ; But not like this!!
*** Illegal Argument.

0007 009F                   Z                    ;Generates 2 bytes as forward ref,
0009 9F                     +Z                   ; Unless qualified.
  009F        Z             EQU       $9F        ;Define Z,
000A 9F                     Z                    ; Now it generates one byte.

  3F01        SWI.GETC EQU        $3F01      ; This feature can be very useful.
000B 3F01                   SWI.GETC             ;Generates SWI followed by code.
  003F        SWI.          EQU       $3F
000D 3F1132                 SWI.,Z,50            ;Generates SWI followed by stuff

0010 FE                     -2                   ;Note: minus sign forces 1 byte
0011 FFFE                   (-2)                 ;Brackets cause default;
0013 FFFE                   #-2                  ;Hash ensures two bytes.

0015 020002                 2,#2                 ;Note: all EXPs can be prefixed
```

SET LOCATION COUNTER

```
        {<LF>}  ORG     <EXP>
```

ORG directs ASM to set the location counter to the value given as the argument. <EXP> is interpreted as a sixteen-bit unsigned value, and may not involve a forward reference.

--Any labels will be EQU'd to the NEW location counter.

--If <EXP> is a forward reference, "Illegal Value" is reported, and the location counter is not changed.

--The value of the location counter will be listed.

Examples:

```
  ØØ1Ø                    ORG     $1Ø      ;New loc. is listed.

  ØØ2Ø          I         ORG     $2Ø      ;Labels are EQU'd...
  ØØ2Ø                    EQU     I        ; ...After the ORG.

  ØØ2Ø                    ORG     J        ;Illegal: J is forward ref.
*** Illegal Value.
ØØ2Ø ØØ2Ø                 FDB     *        ;Note that the ORG
  Ø1ØØ          J         EQU     $1ØØ     ;  wasn't performed.
```

SET DATA PAGE REGISTER

```
        {<LF>}  SETDPR  <EXP>
```

SETDPR directs ASM to assume the DP (Data Page) register has the value <EXP>/$1ØØ (the upper 8-bits of the 16-bit value <EXP>) when attempting to generate Direct Mode addresses. This pseudo-op is only available in the 68Ø9 assembler. The default value selected at the start of each assembly pass is Ø.

--Any labels will be ignored.

--If <EXP> is a forward reference, "Illegal Value" is reported, and the assumed value for the DP is set to zero.

--The address of the page to which DP is set is listed.

Examples:

```
  1FØØ          ORG     $1FØØ
  Ø1ØØ          SETDPR  $123     ;New DP page address listed

  ØØØØ          SETDPR  J        ;ILLEGAL-- J is forward reference
*** Illegal Value
              J         FDB     Ø
```

CONDITIONAL ASSEMBLY DIRECTIVES

ASM supports a powerful set of conditional assembly commands. These commands allow great flexibility in system generation and maintenance especially when multiple configuration or options are needed in a program.

Conditional assembly essentially allows the assembly of only selected portions of the source; portions not assembled are treated effectively as comments. Selection of the desired portions can be specified at assembly time; see "Operator Input Lines".

General features of ASM conditional assembly are:

--Conditional assembly commands may be nested up to a total of 255 levels.

--Source security is enhanced by listing options that can suppress included files, skipped lines and conditional assembly lines.

--Conditional assembly commands are highly consistent: all blocks are terminated by FIN, and all clauses within blocks are separated by ELSE or ELSEIF, depending on function desired.

It should be noted that labels specified on conditional assembly commands are completely ignored.

It should also be noted that forward references are STRICTLY FORBIDDEN as arguments to conditional assembly commands (exception: see IFUND and IFDEF). If an expression is evaluated which contains a forward reference, "Illegal Value" will be reported, and zero will be used as a value.

The following is intended to illustrate use of conditional assembly. Details may be found under the full description of conditional assembly.

SIMPLE CONDITIONAL ASSEMBLY:

```
        IF      <EXP>
        ...
        FIN
```

The expression is evaluated. If it is "true", i.e., positive and non-zero, the lines between the IF and the FIN are assembled; otherwise they are skipped.

ALTERNATIVE ASSEMBLY

```
        IF      <EXP>
        ...
        ELSE
        ...
        FIN
```

The expression is evaluated. If "true", the lines between the IF and the ELSE are executed. If "false", the lines between the ELSE and the FIN are assembled.

NESTED CONDITIONAL ASSEMBLY

```
        IF      <EXP1>
        ...
        IF      <EXP2>
        ...
        FIN
        ...
        FIN
```

The first expression <EXP1> is evaluated. If "false", all lines up to the second FIN are skipped. If "true", lines up to the second IF are assembled, and <EXP2) is evaluated. If "true", the lines in the inner block are assembled; if "false", they are skipped. Then the lines from the first FIN to the second FIN are assembled.

MORE COMPLICATED CONDITIONAL ASSEMBLY

```
        IF      <EXP1>
        ...
        ELSEIF  <EXP2>
        ...
        ELSEIF  <EXP3>
        ...
        ELSE
        ...
        FIN
```

The ELSEIF directive is designed to allow the user to have multiple alternatives without having to nest conditional assembly blocks. Instead of writing,

```
        IF      <EXP1>
        ...
        ELSE
        IF      <EXP2>
        ...
        ELSE
        ...
        FIN
        FIN
```

one can write,

```
        IF      <EXP1>
        ...
        ELSEIF  <EXP2>
        ...
        ELSE
        ...
        FIN
```

which both is clearer and minimizes the possibility of forgetting a FIN. If <EXP1> is "true", then the lines up to the first ELSEIF are assembled, and the remaining lines (up to the FIN) are skipped.

If <EXP1> is "false", then the lines up to the first ELSEIF are skipped. If <EXP2> is "true", then the block of lines between the first and second ELSEIFs is assembled, and the remainder of the lines (up to the FIN) are skipped.

This process of skipping to the next ELSEIF continues until either a ELSE command is encountered or an ELSEIF is encountered with a "true" expression. In either case, all lines up to the next ELSE, ELSEIF or FIN are assembled. ASM then skips to the FIN which closes this block.

MULTI-CASE CONDITIONAL ASSEMBLY

```
        CASE    <EXP>
        ...
        ELSE
        ...
        ELSE
        ...
        ELSE
        ...
        FIN
```

If <EXP> is negative, all lines are skipped to the FIN.

Otherwise, the n-th block of lines is selected to be assembled, and all others are skipped. If <EXP> is zero, the lines between the CASE and the first ELSE is assembled; if <EXP> is one, the lines between the first and second ELSE is assembled; and so on. If <EXP> is greater than the number of ELSE lines in this CASE block, none of the lines are assembled.

CONDITIONAL ASSEMBLY ON UNDEFINED SYMBOLS

```
        IFUND   <SYMBOL>
        ...
        ELSE
        ...
        FIN
```

If the symbol given as an argument is undefined or a forward reference, the block of lines between IFUND and ELSE is assembled; otherwise the block of lines between ELSE and FIN are assembled. The ELSE portion is optional.

This form of the IF directive can be extremely useful for supplying default values to symbols used in controlling system generation. When combined with console input (see "Using ASM", below), it allows certain parameters to be changable at assembly time, without requiring that they be explicitly specified.

Example:

```
                IFUND   BUFSIZE ;BUFSIZE will be
        BUFSIZE EQU     17      ; EQU'd to 17 only
                FIN             ; if it isn't already defined.
```

CONDITIONAL ASSEMBLY ON DEFINED SYMBOLS

```
        IFDEF   <SYMBOL>
        ...
        ELSE
        ...
        FIN
```

This form of the IF directive is the logical opposite of IFUND.

If the symbol given as an argument is defined and not a forward reference, the block of lines between IFUND and ELSE are assembled; otherwise, the block of lines between ELSE and FIN are assembled. The ELSE portion is optional.

Example:

```
                IFDEF   BUFFERSIZE      ;This defines BUFFER
        BUFFER  RMB     BUFFERSIZE      ; only if BUFFERSIZE is defined
                FIN
```

FULL DESCRIPTION OF CONDITIONAL ASSEMBLY COMMANDS

In this section we present a rigorous description of the conditional assembly commands. This section may be skipped by those who do not need to know the exact details of conditional assembly in ASM.

ASM has five states with respect to conditional assembly. Each command will have a different interpretation in each of the five states. These states are:

STATE 1 -- No conditional assembly blocks are being processed. This is the initial state of ASM.

STATE 2 -- Lines are being assembled within a conditional assembly block.

STATE 3 -- Lines are being skipped to the next matching ELSE or ELSEIF.

STATE 4 -- Lines are being skipped to the next matching FIN, i.e., to the end of this conditional assembly block.

STATE 5 -- Lines are being skipped to the n-th matching ELSE.

State one needs no special description. Lines are read and processed unconditionally. When a conditional assembly command is encountered in state 1, ASM switches to one of the other four states, depending on the command and its arguments.

State 2 is similar to state 1. Lines are read and processed unconditionally. Unlike state 1, ELSE, ELSEIF and FIN commands are valid. If an ELSE or ELSEIF command is encountered, ASM enters state 4, skipping to the next matching FIN. FIN commands are ignored unless the FIN closes the outermost conditional assembly block; in this case, ASM switches to state 1.

State 3 skips lines. Lines are read and ignored unless they are one of the conditional assembly directives. If a nested conditional assembly block is encountered, all lines contained in that block are unconditionally skipped. If a FIN is encountered, we go to states 1 or 2 as appropriate. If an ELSE is encountered ASM switches to state 2. If an ELSEIF is encountered, then its argument is evaluated. If "true", ASM switches to state 2, otherwise it remains in state 3.

State 4 is like state 3; the difference is that ELSE and ELSEIF commands are skipped. The only directive that will get ASM out of state 4 is FIN.

State 5 is like state 3, but is used to process CASE blocks. When state 5 is entered, an internal counter records the number of ELSE or ELSEIF lines to be skipped. Lines are processed as for state 4, but when an ELSE or ELSEIF is encountered, the counter is decremented. If the counter then has the value one, ASM switches unconditionally to state 3. An unmatched FIN terminates the CASE block, and ASM switches to state 1.

In the following discussion of the individual directives, the operation of each directive in each state will be described.

Again, note that conditional assembly commands do not allow argument expressions involving forward references.

THE "IF" DIRECTIVE

```
                IF      <EXP>
```

IF is the basic conditional assembly command in ASM.

STATE 1: <EXP> is evaluated. If false (<exp> is zero or negative), ASM transfers to state 3, and skips to an ELSE, ELSEIF or FIN. If true (<exp> is positive and non zero), state 2 is entered.

STATE 2: Exactly like state 1 -- except that if this IF would open the 255-th nested conditional assembly block, "Nesting Error" is reported and the IF is ignored.

STATE 3: ASM unconditionally skips all lines contained in the conditional assembly block which this IF opens.

STATE 4: Same as state 3.

STATE 5: Same as state 3.

CLOSE CONDITIONAL ASSEMBLY BLOCK

```
                FIN
```

FIN marks the end of the most recently opened conditional assembly block.

STATE 1: "Nesting Error" is reported.

STATES 2, 3, 4 and 5: ASM returns to the state it was in when it encountered the line that opened this block.

CONDITIONAL ASSEMBLY WITH CHECK

```
                DO      <EXP>
```

DO's operation is in all cases like IF, with the added restriction that the value of <EXP> is restricted to be +1, Ø or any negative value.

IF SYMBOL NOT DEFINED

```
                IFUND   <SYMBOL>
```

IFUND causes ASM to check the symbol given as the argument. If the symbol given is undefined, the effect is that of IF 1; if it is defined, the effect is that of IF Ø.

From IFUND's point of view, <SYMBOL> is undefined if it is either truly undefined or a forward reference.

STATE 1: If the argument is not solely a symbol, "Illegal Argument" is reported, and ASM switches to state 3. If the argument is solely a symbol, and the symbol is either a forward reference or undefined, ASM switches to state two; otherwise it switches to state 3.

STATE 2: Like state 1, with the additional checks described for IF, state 2.

STATE 3: Same as IF.

STATE 4: Same as IF.

STATE 5: Same as IF.

IF SYMBOL DEFINED

```
                IFDEF   <SYMBOL>
```

IFDEF causes ASM to check the symbol given as the argument. If the symbol given is defined, the effect is that of IF 1; if it is undefined, the effect is that of IF Ø.

From IFDEF's point of view, <SYMBOL> is undefined if it is either truly undefined or a forward reference.

STATE 1: If the argument is not solely a symbol, "Illegal Argument" is reported, and ASM switches to state 3. If the argument is solely a symbol, and the symbol is not a forward reference and is defined, ASM switches to state two; otherwise it switches to state 3.

STATE 2: Like state 1, with the additional checks described for IF, state 2.

STATE 3: Same as IF.

STATE 4: Same as IF.

STATE 5: Same as IF.

THE ELSE DIRECTIVE

```
ELSE
```

The ELSE directive serves to separate groups of lines within a conditional assembly block into clauses.

STATE 1: "Nesting Error" is reported; no other action is taken.

STATE 2: ASM unconditionally changes to state 4, and skips the remaining lines in this conditional assembly block.

STATE 3: ASM unconditionally changes to state 2, and begins assembling lines.

STATE 4: The directive is ignored.

STATE 5: The case counter is decremented. If its value is now 1, ASM switches to state 3; otherwise it remains in state 5.

CONDITIONAL ELSE

```
ELSEIF <EXP>
```

The ELSEIF directive is designed to allow the user to have multiple alternatives without having to nest conditional assembly blocks. ELSEIF is legal anywhere an ELSE is, although in CASE blocks it should be used with caution, as its misuse can lead to difficult code to read.

STATE 1: "Nesting Error" is reported; no other action is taken.

STATE 2: Exactly like ELSE.

STATE 3: It is this state which distinguishes ELSEIF from ELSE. <EXP> is evaluated, and if "true" (positive and non-zero) ASM switches to state 2. Otherwise ASM remains in state 3.

STATE 4: Same as ELSE.

STATE 5: Same as ELSE.

CASE CONDITIONAL ASSEMBLY

```
                CASE    <EXP>
```

The CASE directive causes ASM to select one of the subsequent ELSE-clauses to be assembled.

STATE 1: <EXP> is evaluated. If negative, ASM switches to state 4, and skips all lines in this block. If zero, it switches to state 2, and processes the lines up to the first ELSE. If +1, it switches to state 3, and skips to the first ELSE. Otherwise, it enters state 5, and skips to the n-th ELSE-clause.

STATE 2: Like state 1, with the additional checks described in IF, state 2.

STATE 3: All lines in the conditional assembly block opened by the CASE are skipped.

STATE 4: Like state 3.

STATE 5: Like state 3.

SINGLE LINE ITERATIVE ASSEMBLY

```
        {<LF>}  RPT     <EXP>
```

RPT causes ASM to assemble the next source line zero or more times. If <EXP> is zero or negative, the next line is skipped. If <EXP> is positive, the next line is repeated as many times as specified. <EXP> may not involve a forward reference.

--Any labels present will be EQU'd to the location counter before the line is processed.

--The target of the RPT may not be any of the following directives:

```
      CASE    DO      ELSE      ELSEIF  END     FIN
      IF      IFDEF   IFUND     MON     RPT
```

If such a line is discovered as a target of an RPT, "Nesting Error" is reported, and the line is always processed once.

--Comment lines encountered between the RPT and the next non-comment line will be printed and ignored, NOT repeated.

--If <EXP> involves a forward reference, "Illegal Value" is reported, and the effect is

```
              RPT     Ø
```

--INCLUDE can be the target of a RPT only if the repeat count is Ø or 1.

Example:

```
  ØØØ4          AB      RPT     5       Generate 5 bytes of zero:
ØØØØ ØØ                 FCB     Ø       And EQUs AB to beginning of bloc
ØØØ1 ØØ
ØØØ2 ØØ
ØØØ3 ØØ
ØØØ4 ØØ
  ØØØØ                  EQU     AB      ;Note that AB has right value


                * Generate ascending powers of 2 in 1 byte table:
  ØØØ8          TBL     RPT     8
ØØØ5 Ø1                 FCB     1##(*-TBL)
ØØØ6 Ø2
ØØØ7 Ø4
ØØØ8 Ø8
ØØØ9 1Ø
ØØØA 2Ø
ØØØB 4Ø
ØØØC 8Ø
```

ASSEMBLY CONTROL DIRECTIVES

ASM provides several commands which control various aspects of its operation.

TERMINATE SOURCE FILE

```
                END
                END     <expr>
                MON
```

The END (or MON) directive informs ASM that there are no more source lines to be assembled. When encountered during pass one, it causes ASM to rewind the source file, open the binary and listing files, reprocess the saved operator-input lines (see "Using ASM"), and then process the source file. When encountered during pass 2, it causes ASM to print summaries as requested on the WITH command, to close the source, binary and listing files, and then to exit.

The following summaries may be printed at the end of pass two:

--Symbol table sorted by name and by value

--Line numbers on which errors were detected.

--Number of errors detected.

The last two items will also be printed on the console device.

END or MON need not be present at the end of a source file. If none is found, ASM will supply an END statement.

END statements accept an optional start address expression in the AF field. The expression must evaluate on Pass 2 to a non-zero value. If INCLUDE files are in use, END statements in INCLUDE'd files may set the start address. Multiple ENDs may set the start address if they all evaluate to the same value. Differing values will cause an error. The first value set will prevail.

NOTE: Start addresses of Ø are illegal. This is because the object format uses Ø to indicate NO start address.

SELECT ASSEMBLY OPTIONS

```
        WITH    <OPTION> { , <OPTION> ... }
```

The WITH command provides control over miscellaneous features of ASM. It is intended to be input by the operator at the start of the assembly (see "Using ASM"), but may also be included in the source file. The options in effect at any time are the result of the last WITH command encountered; options not specified are not affected. The options in effect at the beginning of the second pass are the options in effect at the end of the first pass.

The following options are specified by default:

```
        NMCM
        NLN
        NLF
        WI=132
        DE=66
        DMP
        EL
        LST
        DO
```

Options and their meanings are:

MCM

"Motorola compatible". Forces dyadic operators to be evaluated strictly left-to-right. This option should be selected when assembling files originally prepared for assembly by Motorola standard assemblers. Parentheses are still available for overriding Motorola precedence.

NMCM

"Not Motorola compatible". Causes dyadic operators to be evaluated in the usual, heirarchial order. This is the default mode of operation.

M68Ø1

"Assemble M68Ø1 instructions." Accepted only by ASM68ØØ. Tells ASM that M68Ø1 instructions are valid and should be assembled as such. If this option is not enabled, all 68Ø1 specific instructions are treated as M68ØC instructions and 68ØØ equivalent instructions are substituted in their place.

LN

"Line numbers". Informs ASM that there are source line numbers present in the file. See "Source Line Numbers" for a description of how source line numbers are processed.

NLN

"No line numbers". Tells ASM not to expect line numbers in the file. This is the default mode of operation.

LF

"Line feeds". Informs ASM that extraneous line feeds are present in the source file, and are to be ignored. This option should be specified for all files prepared on systems which use CR/LF or LF/CR as line separators. If specified, all line feeds encounterd in the source file will be completely ignored: they will not even be listed.

NLF

"No line feeds". Informs ASM that line feed characters encountered in the file have no special significance. This is the default mode of operation. Files prepared with editors which use CR alone to separate lines should be processed in this mode.

WI=<EXP>

"Set listing device page width". Tells ASM how many physical columns there are on the listing device. Any line that is longer than <EXP> characters long (including assembler generated information such as the output data in the left portion of the page) will be truncated. The default width for a printing device is the page width established by the SDOS SET command; for a disk file, it is defaulted to 132.

DE=<EXP>

"Set listing device page depth". Tells ASM how many physical lines there are on a page. This information is used to control pagination. The default depth for a printing device is the page depth established by the SDOS SET command; for a disk file, the default depth is 66. A depth specification of less than 13 causes continuous form listing (no page headers except the first).

DMP

"Dump symbol table". Requests that a symbol table dump be produced on the listing device at the end of the assembly, sorted by name and by value.

NDMP

"No dump of symbol table." prevents a symbol table dump.

EL

"Save error line numbers". Requests that a summary be printed at the end of the assembly on the listing device, detailing which lines had errors. Each error line saved requires six bytes of memory; if there is no room to save an error line number, no notification is given to the user, and the list of lines at the end of the assembly may not be complete. However, this situation will only arise when ASM has run out of space in the symbol table, which will cause notification of space problems.

NEL

"No error line numbers". Prevents ASM from saving or reporting error line numbers.

LST

"Produce listing". Tells ASM to produce a full listing of the assembled source file. Does not imply DMP.

NLST

"No listing, please". Tells ASM not to produce a full listing of the assembled source file on the listing device. Note that LIST 1 occurring in the source will NOT override this option; note also that any lines on which errors are detected will be listed anyway. This option does not prevent DMP, nor does it imply NDMP.

DO

"Diagnostic output on console". Tells ASM to copy error lines and error messages to the console.

NDO

"No diagnostic output on console". Tells ASM not to copy error lines and messages to the console. This option should be used when a listing is being produced on the console device. Otherwise the output may have intermixed listing and error messages.

THE OPT STATEMENT

```
        OPT     <CHAR>...
```

The OPT command is provided only for compatibility with Motorola source files. The argument field is totally ignored and none of the Motorola specific options are supported. Assembly control directives are specified to ASM via the WITH command.

THE INCLUDE STATEMENT

```
        INCLUDE <FILE NAME>
```

The INCLUDE command allows the inclusion of whole source files as part of the assembly. An INCLUDE'd file is assembled as though its contents were actually substituted for the INCLUDE statement. An INCLUDE'd file may also contain INCLUDE directives. INCLUDE files may be nested in this fashion to a depth of 16 levels.

Common uses for INCLUDE are:

--Inclusion of SDOSUSERDEFS to define SDOS symbols.

--Sharing of common code between several assemblies.

--Breaking up of huge source files to reasonable size files.

The <FILENAME> must be a valid SDOS file name; it is written as though it were a symbol name.

Example:

```
        INCLUDE SDOSUSERDEFS.ASM

        INCLUDE D3:DATATABLES.SRC
```

INCLUDE supports recursion. This means that by use of conditional assembly and recursion, "looping" for multiple copies of a file is possible to the maximum include depth.

Example:

File A:

```
        N       SET     10
        F       SET     1
                INCLUDE FACTORIAL
                FDB     F       ;Gen word with Factorial
                END
```

File FACTORIAL:

```
                IF      N
                FDB     N       ;Generate word with a factor
        F       SET     F*N     ;Compute Factorial
        N       SET     N-1
                INCLUDE FACTORIAL
                FIN
                END
```

When file "A" is assembled it will cause FDB's with values 10 down to 1 and 10! to be generated.

END statements are optional in INCLUDE files. When they are used, they may specify a start address. (See END for more detail).

INCLUDE statements may be the target of RPT statements if the count is 0 or 1. Any other count will produce a "nesting error". Thus a RPT may be used for conditional INCLUDEs.

LISTING CONTROL AND FORMATTING

ASM has an range of listing formatting options, allowing the professional user great flexibility in generating listings which will also serve as documentation.

It is important to understand two underlying design goals of the listing portion of ASM:

--Any line which is determined to be in error will unconditionally be sent to the listing device, regardless of the various listing control options. Such lines will also be sent to the console device, unless supressed by WITH NDO. The lines will be printed on the console exactly as formatted for the listing.

--If WITH NLST was specified (see "Assembly Control"), then ONLY error lines will be listed. In particular, the LIST directive cannot override the NLST option; LIST was designed to be used in formatting listings as they are produced.

--At no time will a totally blank page be printed. Consecutive page-eject commands, or spacing operations which cross multiple page boundaries will never cause more than one page to be ejected; and if encountered at the top of a clean page, they will be ignored.

--There are three listing on/off directives which control whether output goes to the LO (Listing Output). They are listed below in order of decreasing dominance.

```
WITH    LST
WITH    NLST
LIST
```

PAGE HEADING FORMAT

In the following discussions it will be important to know what the various fields of the page header are, and what they are called.

```
   -----------------------------------> Name of Assembler.
   |     -----------------------------> Version of Assembler.
   |     |    ------------------------> PC at Time of Page
   |     |    |                          Eject
   |     |    |       ----------------> Program Name From
   |     |    |       |                  "Name" Command.
   |     |    |       |           ----> Program Title From
   |     |    |       |           |      "Title" Command.
   |     |    |       |           |
   ASM 1.4:   ØØØØ   <NAME>    <TITLE>
<DATE/TIME>; Page 1; Form 1    <SUBTITLE>
<SRC FILE>|    |          |        |
      |   |    |          |        ---> Program Subtitle From
      |   |    |          |              "Page" Command.
      |   |    |          ------------> Count of Control-L's
      |   |    |                         Encountered So Far
      |   |    |                         in file.  Useful
      |   |    |                         when using Editors
      |   |    |                         based on ^L.
      |   |    -----------------------> Page Number, this
      |   |                              listing.
      |   ----------------------------> Date/Time in SDOS
      |                                  System Format.
      --------------------------------> Source File Being
                                         Processed When Page
                                         Throw Occurred.
```

LINE LISTING FORMAT

ASM uses several formats when listing a line. Lines which generate data will be listed in the following form.

```
PPPP DDDDDDDD NNNN: TTTTTTTT....
  VVVV        NNNN: TTTTTTTT....
PPPP          NNNN: TTTTTTTT....
  *S*         NNNN: TTTTTTTT....
              NNNN: TTTTTTTT....
PPPP VVVV     NNNN: TTTTTTTT....
```

The first form is used when a line generates data. PPPP is the first location that the data will be loaded into; DDDDDDDD are the data bytes. Up to four bytes may be displayed on a given line. NNNN is the line number; this field is reset to 1 whenever a form-separator is encountered in the source, and incremented for every line read. TTTTTTT is the text of the source line, with tab characters expanded according to the tab stops currently in effect (specified by TABS).

The second form is used for lines containing directives that do not generate data but do have a numeric result of some kind. VVVV is the value field, and generally is used to display the value of the directive's argument.

The third form is used to list lines containing an ORG command. PPPP is the new value of the location counter.

The fourth form is used to list lines that were skipped due to conditional assembly commands. "*S*" is printed in the value field.

The fifth form is used to list comment lines, and lines which do not have any value per se.

The sixth form is used to list lines containing RMB commands. PPPP is the value of the location counter at the beginning of the reserved block, and VVVV is the number of bytes in the block.

PRINT CONTROL CARDS

```
        PCC     <EXP>
```

PCC instructs ASM as to whether listing control commands are to be printed on the listing. If <EXP> is false, subsequent control commands will not be printed in the listing. If <EXP> is true, subsequent control commands will be printed in the listing.

--PCC is always printed.

--The default at start of assembly is PCC Ø.

TURN LISTING ON/OFF

```
        LIST    <EXP>
```

LIST instructs ASM as to whether subsequent records are to be included in the listing. If <EXP> is true, subsequent lines will be listed; if false, subsequent lines will not be listed.

--LIST has no effect unless a listing is being produced (WITH LST specified).

--PCC controls whether LIST is listed.

--<EXP> may be a forward reference.

--LIST 1 in an INCLUDE'd file will affect the listing of the file containing the INCLUDE directive.

SET TITLE AND EJECT PAGE

```
        TITLE    {;} { <STRING OF CHARACTERS> }
```

TITLE directs ASM to eject a page before printing the next line of the listing. If any non-blank characters occur in the argument field, the title field in the page header is changed before the page is ejected. If no non-blank characters appear, the title is not changed.

--If a semicolon appears as the first non-blank character after the "TITLE" command, it will be ignored, but the title field will be set to whatever characters remain on the line. Thus, a null title may be set by writing

```
              TITLE   ;
```

--The first title specified in pass one will be used as the initial value of the title field at the start of the listing in Pass Two.

--PCC controls whether TITLE is listed.

--If LIST Ø is in effect, the page eject is not issued. However, the title will be changed if a new title was specified.

--If PCC Ø has been specified, then TITLE followed by PAGE will set both the title and the subtitle; only one page will be ejected.

SET SUBTITLE AND EJECT PAGE

```
        PAGE    {;} { <CHARACTERS> }
```

The PAGE directive is identical in function to TITLE, except that it changes the subtitle field of the page header rather than the title field. As with TITLE, the first subtitle encountered in pass one will be used as the initial subtitle of the listing.

SET NAME

```
        NAME    {;} { <CHARACTERS> }
        NAM
```

The NAME directive is similar to the TITLE directive, with the following differences.

--The name field in the page header is affected.

--PCC has no control over the listing of NAME.

--NAME does not eject a page.

The following differences between the Motorola standard and ASM should be noted:

--NAME need not be the first statement in a file.

--More than one NAME directive is permitted.

--No restrictions are placed on the possible contents of the name field.

SPACE LISTING N LINES

```
        SPACE   {<EXP>}
        SPC
```

SPACE directs ASM to insert <EXP> blank lines before printing the next line of the listing. If <EXP> is negative, no lines are spaced; if <EXP> is zero or missing, one line is spaced.

--PCC controls whether SPACE is listed.

--If listed, the SPACE command is listed after the spacing operation has been performed.

--In no case will SPACE cause more spaces to be inserted than remain on the current page.

--If LIST Ø is in effect, no action is taken, although the expression is evaluated.

SET LISTING TABS

```
        TABS    <EXPLIST>
```

TABS allows the user to tell ASM how tabs are to be expanded. Each <EXP> specifies a column number relative to the first column of the source line, which is numbered 1. No <EXP> may be less than 2 or greater that 234. No more than eight tab stops may be specified.

--The default values for tabs are 9, 17, 25, 33 and 41: every eight columns.

--The tabs must be specified in ascending order.

--If any argument to a TABS directive is erroneous, the tabs are reset to their default values.

--PCC controls whether the TABS command is listed.

--At least one <EXP> must be present, and null expressions are NOT allowed.

PRINT SKIPPED RECORDS

```
        PSR     <EXP>
```

The PSR directive allows the user to specify whether records skipped due to conditional assembly commands are to be listed. <EXP> is evaluated; if false, then skipped records are not to be included in the listing; if true, skipped records are to be included in the listing, marked by "*S*" printed in the data field of the listing.

This command can be especially useful when preparing listings of software for release to the users of the software. Code not relevant to the particular system generated can be easily omitted from the listing.

--PCC controls whether the PSR command is listed.

--The default is PSR 1.

PRINT GENERATED DATA

```
        PGEN    <EXP>
```

The PGEN directive is used to tell ASM whether all data generated is to be included in the listing. The expression is evaluated; if false, only the first four bytes of data will be listed. If true, all data will be listed, with additional lines included on the listing if necessary.

--PCC controls the listing of the PGEN directive.

--The default is PGEN Ø.

Examples:

```
  ØØØ1                  PCC     1       ;So we can see the PGEN
  ØØØ1                  PGEN    1       ;So we can see everything.

ØØØØ Ø1Ø2Ø3Ø4           FCB     1,2,3,4,5,6,7,8,9
ØØØ4 Ø5Ø6Ø7Ø8
ØØØ8 Ø9
  ØØØØ                  PGEN    Ø       ;So we only see first part.
ØØØ9 Ø1Ø2Ø3Ø4  L1       FCB     1,2,3,4,5,6,7,8,9
  ØØØ9                  EQU     *-L1    ;It really got generated.
```

PRINT CONDITIONAL ASSEMBLY COMMANDS

```
        PCA     <EXP>
```

The PCA directive is used to tell ASM whether to list conditional assembly command lines. The expression is evaluated; if false, subsequent conditional assembly commands will not be listed. If true, subsequent conditional assembly commands will be listed.

--The default is PCA 1

--PCA controls the listing of the following commands.

```
               DO      ELSE    ELSEIF  FIN
               IF      IFUND   RPT
```

--If a conditional assembly command is being skipped, it will be listed only if PCA 1 and PSR 1 are both in effect.

--PCC controls the listing of PCA commands.

OPERATING ASM

To use ASM, one must first construct a source file containing the text of the assembly language program to be assembled. Under SDOS, this can be accomplished using the context editor (EDIT) or the screen editor (SEDIT). Description of the use of these editors is beyond the scope of this manual.

Once the assembler has started, two kinds of reactions are possible: reactions based upon activities by SDOS on behalf of the assembler (such as opening files, printing on the printer, asking for keyboard data entry, etc.) and actions by the assembler. Interactions with and reactions of SDOS are beyond the scope of this manual; it is assumed the operator is familiar with the SDOS command interpreter, line editing conventions, and how to deal with errors (for more information, refer to the SDOS manual). See Appendix C for a list of some SDOS related error messages that can be reported while using ASM, and their meanings. This manual covers only responses and actions of ASM itself.

STARTING ASM

Once the source file has been constructed, the assembler must be invoked. This is accomplished in the SDOS conventional way, by typing its name while at the SDOS command interpreter prompt; and then entering carriage return. ASM will respond by identifying itself:

```
        .ASM
        Software Dynamics ASM/68Ød, Version 1.4r
```

"d" will be "Ø" if this assembler produces 68ØØ or 68Ø1 object, and "9" if the assembler produces 68Ø9 object code. "r" is the revision letter of ASM.

It will then print

```
        Source File=
```

and wait for input. Enter the name of the file which is to be assembled, followed by a carriage return. If a bad file name is entered, or the file cannot be found or opened, the prompt is typed again, and file name entry is again requested.

After the source file has been successfully opened, the following message will be printed:

```
        Listing File=
```

If a listing is to be produced, the name of the file or device which is to receive the listing should be entered. Otherwise an empty line, indicating "No listing desired", should be entered. If ASM cannot create the file, an error message is printed and the prompt is issued again.

When the listing file has been established, the following message will be printed:

```
        Binary File=
```

If a binary object file is to be produced, the name of the file which is to receive the binary must be entered, otherwise an empty line, indicating "no object file desired", must be entered. If ASM cannot create the file, an error message will be printed and the prompt will be reissued.

OPERATOR INPUT LINES

At this point, all files that need to be opened have been. ASM then issues the following prompt:

```
        >
```

and waits for input. The user now has the option of entering as many valid ASM source lines as desired. The assembler will save them in internal scratch storage and process them at the beginning of each pass (as though they were attached to the front of the source file). This mode can be terminated by inputting an empty line. At that point, the assembler will enter pass one and begin assembling the user's program.

These operator input lines are typically used for two purposes: to enter WITH directives, to establish overall listing or options selection, or to enter EQUate directives to specify values for configuration symbols used by conditional assembly directives embedded in the source file.

It is convenient sometimes to build an SDOS "DO" file that invokes ASM, sets up source, listig and object files, and then specifies (as operator input lines) a specific set of configuration parameters for the program being assembled. The DO file then "represents" a particular configuration of the assembled program, and can be used to easily regenerate that configuration. Other DO files would represent other configurations.

Examples of use:

1) The following procedure allows the user to assemble a simple program typed in from the console.

```
ASM VERSION 1.x / xxxx
SOURCE FILE=CONSOLE:
LISTING FILE=MYLIST
BINARY FILE=MYBIN
>       ORG     $100
>       LDX     #$FE00
>L1     CLR     0,X
>       INX
>       CPX     #$FEFF
>       BNE     L1
>       RTS
>       END
```

The program will be assembled, and a listing produced. Note that operator input was terminated not by an empty line, but by ASM discovering the END directive.

2) Assemble a file called "PROCESS.ASM", generate no listing and no binary. This can be useful when checking for errors. (The file name extension ".ASM" is used purely by convention; any valid SDOS filename can be used here).

```
ASM VERSION 1.x / xxxx
SOURCE FILE=PROCESS.ASM
LISTING FILE=
BINARY FILE=
>
```

3) Assemble "PROCESS.ASM", generate no listing, but do generate a symbol table dump in on the printer device. The listing format used will match the printer's width and depth as specified by the SET program.

```
ASM VERSION 1.x / xxxx
SOURCE FILE=PROCESS.ASM
LISTING FILE=LPT:
BINARY FILE=
> WITH NLST,DMP
>
```

4) Assemble "PROCESS.ASM", generate a listing in file "PROCESS.LPT" and specify listing page sizes, overriding the default used with files.

```
ASM VERSION 1.x / xxxx
SOURCE FILE=PROCESS.ASM
LISTING FILE=PROCESS.LPT
BINARY FILE=
> WITH WI=105,DE=51
>
```

5) Assemble "PROCESS.ASM", generate listing as before, generate binary, specify listing format, and define certain symbols that control conditional assembly of PROCESS.

```
ASM VERSION 1.x / xxxx
SOURCE FILE=PROCESS.ASM
LISTING FILE=PROCESS.LPT
BINARY FILE=PROCESS.BIN
> WITH WI=105,DE=51
>MEMSIZE EQU    $4000
>PROGBASE EQU   $1000
>USEFLOPPY      EQU     1
>
```

ERROR MESSAGES

If an error is detected while processing a source file, the following actions are taken:

--The line and the error messages are listed on the console, unless WITH NDO was specified. A printed error message refers to the line printed immediately preceding it.

--The line and the error messages are listed on the listing device if one was specified.

The following error messages can be reported:

*** Double Definition.
The line contains an attempt to define a symbol whose value is also defined elsewhere. Only the first definition is honored.

*** End of Source File Encountered.
The end of the source file was encountered before an END command was seen.

*** Illegal Argument.
A bad argument field was detected. Examples:

```
Y       EQU
        ORG
```

*** Illegal Digit.
Indicates the presence of a malformed number. Examples:

```
34F
$1HJ
%LMN
```

*** Illegal Label.
An illegal label was detected. Either the label field contains a non-symbol, or a label was specified on a line without a command.

*** Illegal String.
Indicates an attempt to define a string that included a carriage return.

*** Illegal Value.
Usually indicates that a forward reference was present where forward references are not allowed (see appendix A).

*** Impossible forced reference (<< or <).
Use of "<<" with ",PC", ",PCR" or "[...]" or "<" used with DPR value not correctly set.

*** INCLUDE file not found.
INCLUDE file was not found. INCLUDE was ignored.

*** Input Line Too Long
Input line was too long and was truncated prior to processing.

*** Nesting Error.
The line violates some rule regarding condtional assembly. Possibilities are:

--ELSE, ELSEIF or FIN encountered when no conditional assembly block was open.

--IF, CASE, IFUND or DO encountered that would cause conditional assembly commands to be nested more than 255 deep.

--The argument of an RPT command was a command which cannot be RPT'd.

*** Out of Memory.
Indicates that there was no room to enter a new symbol into the symbol table. Assembly continues, but the symbol will remain undefined throughout the assembly. No more error line numbers will be saved.

*** Phase Error.
Indicates that the assembler has been asked to give a symbol a different value in pass two than it was given in pass one. Usually caused by "Out of Memory."

*** Register Field Missing
Indicates that a machine instruction which requires a register was written with an undecipherable register field.

*** Start address =Ø or does not match other end(s).
Start address specified has value zero (illegal under SDOS) or multiple ENDs specified differing start addresses (Multiple ENDS are possible when INCLUDE is used).

*** Syntax Error.
A malformed expression or addressing mode was encountered. Examples:

```
3+
(
3?7
ADDB    O,S+
```

*** Undefined Symbol.
The line contains a symbol that was not defined. This can occur with symbols which involve more than one level of forward reference.

*** Use of Doubly-defined Symbol.
The line contains a use of a doubly-defined symbol.

SYMBOL TABLE DUMP FORMAT

This section describes the format of ASM symbol table dumps. The symbol table is dumped sorted by name and by value, using a common format for the symbol entries:

```
Qname/Ø123
| |     |
| |     ----->  This field contains the symbol's
| |               value.  If the symbol is undefined,
| |               this field is "****".
| |
| ---------->  This field contains the symbol's name.
|
------------>  This field contains a qualifier
                 which gives additional
                 information about the symbol.
               Possibilities are:
                 "*"   Indicates unused symbol.
                 "+"   Indicates doubly-defined symbol.
                 Blank indicates none of the above.
```

ERROR LINE SUMMARY FORMAT

At the end of pass two, after the symbol table dump, ASM will print out a list of lines on which errors occurred. Each item in the list has the following format:

```
        ffff-llll
```

"ffff" specifies the Form number that contained the line, as printed in the Form field of the page header. "llll" specifies the line within that form which was in error, as listed in the line number field of the listing. This is especially useful when using EDIT 1.1, as the offending line can easily be found by telling the editor to do a "EBfilename\ffffEYllllJ", which goes to the ffffth form, llllth line.

MEMORY USAGE AND SIZING CONSIDERATIONS

ASM dynamically allocates memory at runtime for the following kinds of data:

--User-defined symbols require at least 7 bytes of storage. The amount used by a given symbol will be
6 + LEN(SYM)
where LEN(SYM) is the number of characters in the symbol name. Only the first thirty-two characters of a symbol name are saved.

--Operator-input lines require an amount of space that varies with the length of the line. The formula is:
3 + LEN(LINE)
where LEN(LINE) is the number of bytes in the line, including carriage return.

--Saved error line numbers require 6 bytes of space each.

ASM will automatically use all the memory between the end of ASM and the top of the user space (see SDOS manual).

It is recommended that ASM be run on a system with at least 16K bytes of user space. This will allow approximately 4K bytes of usable space for ASM runtime tables.

APPENDIX A -- ASM Directives which Disallow Forward References:

The following directives will not allow the user to use forward references in their argument lists:

CASE
DO
ELSEIF
IF
ORG
PCA
PCC
PGEN
PSR
RMB
RPT
SET
SETDPR
SPACE
TABS

APPENDIX B -- ASCII Character Set

	$ØØ	$1Ø	$2Ø	$3Ø	$4Ø	$5Ø	$6Ø	$7Ø
$Ø	NUL	DLE	BLANK	Ø	@	P	`	p
$1	SOH	DC1	!	1	A	Q	a	q
$2	STX	DC2	"	2	B	R	b	r
$3	ETX	DC3	#	3	C	S	c	s
$4	EOT	DC4	$	4	D	T	d	t
$5	ENQ	NAK	%	5	E	U	e	u
$6	ACK	SYN	&	6	F	V	f	v
$7	BEL	ETB	'	7	G	W	g	w
$8	BS	CAN	(	8	H	X	h	x
$9	TAB	EM	)	9	I	Y	i	y
$A	LF	SUB	*	:	J	Z	j	z
$B	VT	ESC	+	;	K	{	k	{
$C	FF	FS	,	<	L	\	l	|
$D	CR	GS	-	=	M	}	m	}
$E	SO	RS	.	>	N	^	n	~
$F	SI	US	/	?	O	_	o	RO

APPENDIX C -- Common I/O Error Messages

ERROR	MEANING
1Ø11	Can't find file
1Ø15	Disk space exhausted
1Ø23	File name doesn't start with A-Z or $
1Ø34	Illegal Device operation requested

Error 1Ø34 generally indicates that an output-only device has been specified as the source file, or that an input-only device has been specified as the listing or binary file.

APPENDIX D - 680C Compatibility Instruction Set

The 680C is an imaginary processor whose instruction set includes most of the 6800, 6801, 6802, and 6803 instruction sets (the exceptions are those instructions dealing with processor context). Many of the 6809 functions are supported by the 680C. In addition, the 680C instruction set includes some instructions which don't exist on any of the real 680x processors, but implement frequently-used code sequences. Note: the 6805 is NOT covered by 680C.

The value of the 680C is that code written using its instruction set will execute on any of the 680x processors, although more efficiently on some than on others. This means that some features of the more advanced processors may be used, while preserving backward compatibility with earlier processors through emulation of the advanced instructions.

The ASM6800 and ASM6809 assemblers both accept 680C instructions. ASM6809, of course, also accepts the balance of the 6809 instruction set. The emulation feature of the 680C assembler is a two-edged sword: on the one hand, it offers a consistent instruction set across the family of 680x processors; on the other hand, emulated instructions often have side-effects of which the coder should be aware. Overall, the value gained from a consistent instruction set outweighs the constraints introduced by the emulation of instructions.

In certain cases, the emulation constraints are untenable. The coder then has recourse to conditional assembly:

```
IF      M6800
...
ELSEIF  M6801
...
ELSEIF  M6809
...
FIN
```

to produce code sequences optimized for a particular processor.

Note that the meaning of a 680C istruction is the intersection of the meanings of the implementations of the 680C instruction on all processors; this generally means that many 680C instructions leave the condition code bits in an undefined state. Other side effects are also possible, such as damage to X register contents or the memory location TEMPX.

The 680C instruction set includes nearly all of the 6800/6801 instructions, a good portion of the 6809 instruction set (replete with many of the more popular addressing modes), and a few new instructions added, due to popular demand.

First, all 6800 instructions except the following are included in 680C:

TAP	TPA	WAI

The following 6801 instructions are included in 680C (thus allowing their use on the 6800):

ABX	ADDD	ASLD	BHS	BLO
LDD	LSLD	LSRD	MUL	PSHX
PULX	STD	SUBD		

The entire set of 6809 long conditional branches (except LBRN) are included in 680C.

For the instructions

ADCD	ADDD	ANDD	CMPD	DECD
EORD	INCD	LDA	LDB	LDD
ORA	ORB	ORD	SBCD	STA
STB	STD	SUBD		

most of the 6809 addressing modes may be used. The ones that may not be used are:

accumulator-offset indexed

program counter relative

indexing using registers U, Y, PC or PCR

Additional limitations are:

constant offsets may be positive only, and must be in the range $00 - $FF

auto post-increment is not allowed in the indirect addressing mode

The following 68Ø9 instructions are implemented in a limited fashion:

```
                LEAX    k,X
                LEAS    k,S
```

where -16 <= k <=15. Note that the destination and index registers must be the same for each instance of the instruction.

The 68Ø9 instructions

```
                TFR     A,B
                TFR     B,A
```

are allowed. No other TFR class instruction is allowed. Note that 68ØC TFR leaves the CC bits (except carry) undefined. TAB and TBA can be used if setting the CC bits is desired, but TFR is faster otherwise.

The 6809 instruction CMPD, subject to the addressing and conditional branch restrictions above, is included in 680C. Only the instructions

BEQ	LBEQ	BNE	LBNE	BCC
LBCC	BCS	LBCS	BHI	LBHI
BLS	LBLS	BLO	LBLO	BHS
LBHS				

may immediately follow the CMPD instruction.

The instructions

JMP	JSR

are allowed, but cannot have auto post-increment of any kind, in any addressing mode.

The following 6809 instructions are not allowed. All instructions using the U or Y registers are implicit members of this list:

ANDCC	BRN	CMPS	CWAI	EXG
LBRN	LBSR	ORCC	PSHS	PSHU
PULS	PULU	SEX	SWI2	SWI3
SYNC				

Instructions peculiar to the 68ØC (and their equivalent expansions) are:

```
NEGD                            COMD
        NEGA                            COMB
        NEGB                            COMA
        SBCA    #Ø

ORD                             EORD
        ORAB    arg+1                   EORB    arg+1
        ORAA    arg                     EORA    arg

INCD    arg                     DECD    arg
        INC     arg+1                   TST     arg+1
        BNE     x                       BNE     x
        INC     arg                     DEC     arg
  x:                              x:    DEC     arg+1

OKRTS                           ERRORTS
        CLC                             SEC
        RTS                             RTS

ROLD                            RORD
        ROLB                            RORA
        ROLA                            RORB

TDX                             TXD
        STAA    TEMPA                   STX     TEMPX
        STAB    TEMPB                   LDAA    TEMPA
        LDX     TEMPX                   LDAB    TEMPB

TDS                             TSD
        TDX                             TSX
        TXS                             TXD

ASRD                            SBCD    arg
        ASRA                            SBCB    arg+1
        RORB                            SBCA    arg

ADCD    arg                     ANDD    arg
        ADCB    arg+1                   ANDB    arg+1
        ADCA    arg                     ANDA    arg

BNED    arg                     BEQD    arg
        IF      \M68Ø9                  IF      \M68Ø9
        BNE     arg                     BNE     x
        TSTB                            TSTB
        FIN                             FIN
        BNE     arg                     BEQ     arg
                                  x:

PULD                            PSHD
        PULA                            PSHB
        PULB                            PSHA
        TSTD                          TSTD
                TSTB    68ØØ/68Ø1       SUBD    #Ø 68Ø9
                BNE     *+3
                TSTA
```

NOTE: The instructions BEQD and BNED may be used only immediately following:

ORD	EORD	ADDD	SUBD	ADCD
SBCD	LDD	STD	ASLD	ROLD
COMD	NEGD	ANDD		

Side effects to watch for using 68ØC:

The side effects should be "obvious" if careful thought is given to the problem of making 68ØC code work on any of the 68Øx processors.

The contents of the X register will be undefined after execution of a 68ØC instruction that uses S as an index register (except LEAS), or uses indirection ("[" "]") in an addressing mode.

Use of auto-increment in an instruction leaves the condition codes in an undefined state.

The MUL instruction will alter the location TEMPX (**).

The PSHX instruction will alter the B register and the location TEMPX (**).

The TXD instruction will alter location TEMPX (**).

The ABX instruction will alter the B register and the location TEMPX (**).

The TDX instruction will alter location TEMPX (**).

The CPX instruction will alter the carry condition.

The TFR instruction will alter all arithmetic conditions, except carry.

The memory locations $ØØ - $18 do not exist on the 68ØC, as this would be incompatible with the 68Ø1.

Double-register instruction operations are not indivisible with respect to interrupts.

(**) These instructions should NEVER be used in interrupt service routines. The instruction sequence interrupted may use TEMPX.

Sample 68ØC code (works on 68ØØ, 68Ø1 and/or 68Ø9:)

```
* Search BUFFER in blocks of 4 for word TARGET
SEARCHBUFFER
        LDD     WORD               ; Note we can do this on 68ØØ!
        LDX     #BUFFEREND+2
SEARCHBUFFERLOOP
        LEAX    -2,X
        CMPX    #BUFFERBASE        ;buffer searched?
        BEQ     SEARCHBUFFERFAIL           ;b/ yes, didn't find it
        CMPD    ,--X               ; CMPD ,X+++ would destroy CC bits
        BNE     SEARCHBUFFERLOOP           ; note use of BNE, not BNED here!
        RTS                        ; assent: carry is reset

SEARCHBUFFERFAIL                   ; signal fail: exit with carry set
        ERRORRTS
```

NOTE: When the target processor of an assembly is a 6800, the MUL instruction is emulated by emitting a

```
          JSR     MUL6809
```

instruction. The user is responsible for supplying that code -- preferably using conditional assembly:

```
          IF      M6800
          .
 *        MUL6809 -- Subroutine to simulate 6809 style
 *                    "MUL" instruction
 *
 MUL6809  STAA    TEMPA   Save multiplicand
          RORB            Look at first multiplier bit
          BCS     *+3     B/ 1st bit is one!
          CLRA            1st mult. bit is 0, set part. prod.
          LSRA            Perform multiply iteration
          RORB
          BCC     *+4
          ADDA    TEMPA
          RORA
          RORB
          BCC     *+4
          ADDA    TEMPA
          RORA
          RORB
          BCC     *+4
          ADDA    TEMPA
          RORA
          RORB
          BCC     *+4
          ADDA    TEMPA
          RORA
          RORB
          BCC     *+4
          ADDA    TEMPA
          RORA
          RORB
          BCC     *+4
          ADDA    TEMPA
          RORA
          RORB
          BCC     *+4
          ADDA    TEMPA
          RORA
          RORB
          RTS
          FIN
```

INDEX

INDEX

INDEX