ASM09
6809 OPTIMIZING ASSEMBLER
VERSION 3.2

USER'S GUIDE

IMPORTANT NOTE

Although every effort has been made to make the supplied software and its documentation as accurate and functional as possible, Southwest Technical Products Corporation will not assume responsibility for any damages incurred or generated by such material. Also, Southwest Technical Products Corporation reserves the right to make changes in such material at any time.

SOUTHWEST TECHNICAL PRODUCTS CORPORATION
219 W. RHAPSODY SAN ANTONIO, TEXAS 78216

ASM09

6809 OPTIMIZING ASSEMBLER

VERSION 3.2

USER'S GUIDE

IMPORTANT NOTE

Table of Contents

PREFACE

This publication was designed as a reference manual for the SWTPC 6809 Optimizing Assembler, Version 3. It is not intended as a tutorial on assembly language programming, nor is it intended as a reference on the 6809 microprocessor. Although detailed descriptions are provided for the native 6809 instructions, these descriptions should not be considered exhaustive. The Motorola MC6809 Programming Manual should be consulted for more information on the 6809 microprocessor. For a tutorial introduction to assembly language programming, the book COMPUTER ORGANIZATION AND PROGRAMMING by W. Gear is an excellent choice.

1.0 - Southwest Technical Products Assembler

The SWTPC 6809 resident assembler is a very powerful disk assembler designed to provide a versatile programming tool. It has many special features added to support structured programming techniques and enhance code modularity and readability. In addition, the assembler provides a multi-pass optimizer that attempts to reduce the size and execution time of assembled object code. Two options are provided to selectively disable certain types of optimization in order to reduce the time required for an assembly.

1.1 - Required Environment

The assembler runs on a Southwest Technical Products 6809 microcomputer system running the FLEX operating system. A minimum of 16K of user memory is required (implying 24K total memory) and provides approximately 4.8K of symbol table space. The assembler does not support a virtual symbol table, so that in systems with limited memory, it is possible to overflow the symbol table with very large programs. The assembler requires approximately (8.4 + Ls) * Ns bytes of symbol table space, where Ls is the average number of characters in a symbol and Ns is the total number of symbols to be kept in the dictionary.

Both qualified data structures and procedures require larger (48 byte) entries in the symbol table, so that the use of many structures and procedures will somewhat reduce the amount of available symbol table space. Similarly, each level of library inclusion requires a buffer area (336 bytes) in the symbol table, and will also reduce the amount of available symbol table space. Note that this space is required for each level of inclusion, not for each inclusion file processed.

1.2 - Assembler Distribution

The ASM09 program is distributed on Flex format 5-inch and 8-inch floppy disk. The disk contains the Flex operating system, the "CAT", "COPY", "NEWDISK", and "LINK" utility programs (to enable duplication of the disk), and the four supplied assembler files:

"ASM09.CMD"	The main assembler command file.
"ASM09.CMV"	The assembler symbol table overlay.
"MIKCV.CMD"	The binary file to Mikbug format converter.
"BINCV.CMD"	The Mikbug format to binary file converter.

The assembler command file and overlay may be renamed as long as the command file and the overlay file are given the same name. For example, if the command file were named "ASM.CMD", the overlay name would then be "ASM.CMV". If the overlay file is not renamed, the assembler will be unable to produce an address file or a symbol table listing.

1.3 - Assembler Command Syntax

The general syntax of the ASM09 command is:

```
+++ASM09 <input file> [,<output file>] [,+<option list>]
```

The first file specification is the name of the file to be assembled. This file specification is required. The second file specification is the name of the binary file to be generated by the assembler. If no output file is specified, its name defaults to that of the input file. If the output file extension is not specified, it defaults to ".BIN". If the specified output file already exists on disk, the old file will be automatically deleted and replaced by the new file. Assembler options are specified on the command line by placing them after a plus sign, to separate them from file specifications.

1.4 - Assembler Options

The option list consists of single characters, optionally separated by commas, and terminated by a carriage return or FLEX end of line character. The options that may be specified are listed below:

A - Generate Address File. The "A" option will cause the assembler symbol table overlay to generate an external symbol address file. Only those symbols defined as entry points to the global dictionary will be included.

B - Suppress binary output. The "B" option will suppress generation of a binary output file. If the binary file already exists on disk, it will not be deleted.

C - Suppress Cautions. The "C" option will suppress all caution messages produced by the assembler.

E - Suppress Error Messages. The "E" option will suppress all error, warning, and caution messages produced by the assembler. Since all diagnostic messages produced by the assembler are suppressed, it is possible that errors in the source program being assembled will go undetected by the user.

F - Optimize Assembly Time. The "F" option will cause the assembler to suppress any optimization of object code. Foreward references will be assembled using the least restrictive addressing modes. This option will force the assembler to complete in two passes, but object code may be considerably larger than required. This option is especially useful while debugging a program which will later be optimized. Note that the "R" option takes priority over this option in the determination of branch lengths.

G - Enable generated code output. The "G" option will cause the assembler to print all generated binary object code. If this option is not specified, the assembler will print up to eight bytes of object code on the same line as the source statement and then suppress any additional printed output. Note that this option does not affect the binary file produced.

L - Suppress listing. The "L" option will suppress any printed output from the assembler, except for lines containing errors detected by the assembler.

M - Specify Motorola Compatability. The "M" option will supress non-Motorola extended processing. Index addressing optimization is suppressed and branch range checking is selected. All labels are internally truncated to six characters of significance. Arithmetic expressions are evaluated using a strict left-to-right order. Character constants revert to the single quote only Motorola format. If Mikbug format object code is desired, the assemblers binary output may be converted using the MIKCV utility program.

N - Suppress Line Numbers. The "N" option will cause the assembler to suppress line number output. This option can be used to reduce the size of the assembler listing.

P - Format Page Output. The "P" option will cause assembler output to be formatted for a printer. The assembler will ask for a heading for the assembly, and perform page counting and title functions. If this option is not specified, the PAGE and TITLE mnemonics are ignored by the assembler.

R - Suppress Branch Range. The "R" option will cause the assembler to suppress branch/long branch optimization. The assembler normally treats branch and long branch mnemonics as identical, and computes which type of branch is required. If this feature is suppressed, branches are limited to approximately 127 bytes range, and an error message will be produced if a range error is detected. This option will normally reduce the number of optimization passes required.

S - Suppress symbol table. Whenever the assembler produces an object code listing, it normally produces a sorted, formatted listing of its symbol table. The "S" option suppresses this output. It is not possible to produce a symbol table listing without producing an object code listing.

T - Truncate Print Output. The "T" option will reduce the number of bytes of object code per line in order to decrease the width of the output listing. Together with the "N" option, the output width is decreased sufficiently to obtain printouts on an eighty column printer.

U - Print Unnamed Dictionaries. The "U" option will cause the assembler to print unnamed procedures (i.e., procedures with a name of "*PRnnnn") found in the symbol table. Unnamed procedures are normally procedures included from system library files and are of marginal value in the symbol table listing.

W - Suppress Warnings. The "W" option will suppress all warning and caution messages produced by the assembler.

1.5 - Conversion Programs

The output of the assembler program is a binary file in a format suitable for the system loader. This format is compact and efficient, but it is not compatable with the Mikbug paper tape format object code required by several prom monitors and cassette tape interfaces. Two utility programs have been provided to convert from binary format to Mikbug format and conversly.

The MIKCV program converts a binary file (such as is output from the assembler) into a text file in Mikbug format. Similarly, the BINCV program will convert a Mikbug format text file into a binary file, at a significant savings in disk space and execution time. The syntax of the two commands is as follows:

```
+++MIKCV <input file>, <output file>

+++BINCV <input file>, <output file>
```

In each case, both the input file specification and the output file specification are required. For MIKCV, the input file must be a binary file (an extension of .BIN is assumed) and the output file must be a text file (a .TXT extension is assumed). For the BINCV program, the input file must be a text file (a .TXT extension is assumed) and the output file must be a binary file (with an assumed extension of .BIN). If the output file exists on disk it will automatically be deleted and replaced with the new output file.

In general, Mikbug format files will be approximately 2.7 times larger than the equivalent binary file. Mikbug format files output records of 16 bytes maximum, and include a transfer address in the header block. The name of the output file is used as the name placed in the Mikbug header by MIKCV. The name placed in the Mikbug header is ignored by the BINCV program.

2.0 - Input Language Syntax

Input to the assembler consists of one or more disk files. These files are expected to be in 8-bit ASCII code, with the sign bit always set to zero. These files may be space compressed, and may contain control characters. The assembler treats carriage returns, form feeds, and rubouts as input line delimiters. The horizontal tab character is treated as a white noise character (same as a blank). All other control characters are ignored by the assembler.

2.1 - Character Classifications

Each character in the input stream is classified into one of four groups: alphabetic type characters, numeric type characters, special characters, and separator characters (white noise).

Alphabetic type characters consist of both upper and lower case letters, the underbar character, and the backslash character. In general, the assembler will make a distinction between upper and lower case letters in symbols defined by the user, but will not make that distinction for symbols defined internally to the assembler. For example, user labels "label" and "LABEL" are separate and distinct, while the register name "IX" is identical to "ix", or for that matter, "Ix" or "iX". This permits the assembler to be used in either upper or lower case environments with a maximum of compatability.

Numeric type characters consist of the digits zero through nine, the crosshatch "#", the dollor sign "$", the question mark "?", and the at sign "@". It is important to realize that numeric type characters are not necessarily digits. When the assembler is recognising a number, the digits in the number start at zero and continue to one less than the number's radix. In the case of hexadecimal numbers, the letters A through F are considered digits, even though thay are alphabetic type characters.

Special characters are used by the assembler as comment or conditional assembly designators, quoted string delimiters, and as operators. Since the function of special characters depends heavily upon the context in which they are encountered, they are best documented along with the functions they perform.

Separator characters consist of the horizontal tab character, the carriage return character, the rubout character, and the space character. These characters serve to separate assembler tokens and in general, have no significance themselves. A special separator character is the semicolon, ";", which besides being a white noise character, is used to denote the presence of secondary assembler source statements.

2.2 - Identifiers

Identifiers consist of a leading alphabetic type character, followed by one or more alphabetic or numeric type characters. The maximum length of an identifier permitted by the assembler is 127 characters, while the minimum length is two characters. Note that while single-character identifiers are not expressly prohibited, defining identifiers with names like "A" or "B" can lead to unexpected results when using indexed addressing modes. In general, it is considered good coding practice to use identifiers with names that are contextually meaningful instead of identifiers with arbitrary and meaningless names.

In certain cases, special terminating characters may be used to denote the end of an identifier, for example, the qualified reference "EMPLOYEE.PENSION" contains the period character as a terminator of the identifier "EMPLOYEE". The terminating character is not considered as part of the identifier; it is considered as part of the qualified reference. It is important to understand that characters like the underbar and the backslash are valid identifier characters and are not equivalent to special purpose terminator characters. The assembler treats all characters of an identifier as significant.

Some examples of valid identifiers are:

```
MONTH
THIS_IS_A_VERY_LONG_IDENTIFIER
\Break\
lower_case_identifer
task_done?
PLM$STYLE$IDENTIFIER
```

If the Motorola compatability option has been selected, the assembler will internally truncate all identifiers to six characters. If the identifier was originally longer than six characters, the excess is simply discarded. No warning is issued unless the truncation process results in multiple definitnions of a single identifier.

2.3 - Implicitly Defined Identifiers

Assembler initialization places several identifers in the global dictionary and assigns their values prior to beginning the first pass on the input source file. These identifiers are protected symbols, i.e., any attempt to redefine their value will result in an error message, with no change in in the identifier value. These implicitly defined identifiers are as follows:

DAY --	The current day of the month, as two ASCII characters in a sixteen bit value with the star attribute set.
FALSE --	The truth value "FALSE".
MONTH --	The current month of the year, as two ASCII characters in a sixteen bit value with the star attribute set.
TRUE --	The truth value "TRUE".
YEAR --	The last two digits of the current year, as two ASCII characters in a sixteen bit value with the star attribute set.

2.4 - Input Statements

The assembler input consists of one or more files containing assembler language source statements, assembler directives, and comment statements. Source statements assemble into actual machine code instructions, and in general have a one to one corespondence with machine operations. Assembler directives set environmental parameters affecting machine code generation, listing format, and dictionary structure. Comment statements are used to document and format an assembler program listing but are otherwise unprocessed by the assembler.

2.5 - Comment Statements

Comment statements begin with either a plus sign "+", an asterisk "*", or a period ".", and are terminated with a carriage return character. If pagination is selected via the assembler "P" option, the plus sign and asterisk type comments assume a special significance. Those comments beginning with a plus sign force the assembler to a new page, as if a PAGE mnemonic had immediately preceded the comment. Comments beginning with an asterisk cause the assembler to force a new page if fewer than 14 lines remain on the current page. This facility is extremely convenient for preventing logically connected sections of source code from overflowing page boundaries.

2.6 - Source Statements

The assembler classifies source statements into primary and secondary statements according to their position in an input line. Primary statements begin in column one of the input line, and are terminated by a carriage return or a semicolon. If the primary statement was terminated with a semicolon, one or more secondary statements may follow it on the same line, each terminated by a semicolon or carriage return. The only restriction on statement format is that optional fields, if present, must start by column 30 of the input record, or be separated from the preceeding field by only one space.

2.6.1 - Primary Statements

Primary statements consist of an label field, an mnemonic operation code field, an operand field, and an comment field. All statement fields are optional however, an operation code field must be present if an operand field is to be used. Some operations have restrictions on label and/or operand fields. Notice that a null line is a valid primary statement, as is a line with only a comment (which must begin after input column 30).

Labels, if present, must begin in column one of a primary statement, and consist of a valid assembler identifier. The label should terminate with a space, a period, or a colon. Labels terminating with a colon are defined in the parent dictionary of the current dictionary and represent explicitly declared entry addresses. Labels terminating with a period are defined in the global dictionary and represent global definitions. The value of a primary source statement label is the value of the program counter at the beginning of statement evaluation, and has the relocation attributes of the currently active program counter.

In order to clear up some of the details of the previous paragraphs, an example of assembly source statements is provided. This section of code is a subroutine to perform a single bit right arithmetic shift on a multi-byte field.

```
                    1.  *
                    2.  .    SUBROUTINE TO SHIFT A FIELD ONE BIT RIGHT
                    3.  .
                    4.  .    ENTER WITH X => FIELD TO SHIFT
                    5.  .                B =  BYTE COUNT OF FIELD
                    6.  .
                    7.           PROC
0000  A6 84         8.  SHIFT:   LDA   0,X      GET FIRST BYTE
0002  47            9.           ASRA           SHIFT RIGHT ARITHMETIC
0003  A7 80        10.           STA   0,X+     PUT BACK IN MEMORY
0005  5A           11.           DECB           DECREMENT THE BYTE COUNT
0006  A6 84        12.  ROTATE   LDA   0,X      GET NEXT BYTE
0008  46           13.           RORA           ROTATE RIGHT ONE BIT
0009  A7 80        14.           STA   0,X+     PUT BACK IN MEMORY
000B  5A           15.           DECB           DECREMENT BYTE COUNT
000C  26 F8        16.           BNE   ROTATE   CONTINUE TILL DONE
000E  39           17.           RTS
                   18.           END
```

Lines 1-6 of the above subroutine are comments explaining what the routine does and how it is to be parameterized. Such comments are, strictly speaking, unnecessary in that the assembler ignores them. They are provided to benefit programmers (perhaps yourself) attempting to understand the code. It is always considered good coding practice to type a few extra lines to thoroughly document subroutines. Notice that line 1 is an asterisk-type comment. This line helps to assure that this routine will not cross over a page fold in a printed listing.

Lines 8-17 are assembler source statements and represent actual 6809 machine instructions. The object code generated by the assembler appears to the left of the line number. The label "SHIFT:" on line 8 is an explicitly declared entry point to the subroutine and has the value 0000 (Hex). The "LDA" in line 8 is an assembler mnemonic for "Load Accumulator". The "0,X" is the operand and signifies indexed addressing mode (see addressing modes). The label "ROTATE" on line 12 is a local label and has the value 0006 (Hex). It is not defined anywhere outside of the subroutine and will not conflict with other similarly named labels.

Lines 7 and 18 are assembler directives and are used to delimit the subroutine. For more details on their function, consult the chapter on assembler directives.

2.6.2 - Secondary Statements

Secondary statements begin after the semicolon terminating a primary statement, and consist of a mnemonic operation code field, an operand field, and a comment field. Labels may not be defined in a secondary statement. As before, operand and comment fields are optional. Note that operation codes are required in secondary statements. If the Motorola compatability option has been selected, no secondary statements will be recognised. A short segment of code containing secondary statements will serve to illustrate their utility:

```
        LDD     CHEKSUM             Pick up the Checksum
        LSRA; ROLB                  Shift it Left one Bit
        ADDD    NEWORD              Add in the Next Word
        STD     CHEKSUM             Stuff Back in Sum
```

2.6.3 - Mnemonic Operation Codes

Assembler source statement mnemonic operation codes may be 6809 operation mnemonics listed Table 2.1, 6800 Family compatability mnemonics, listed in table 2.2, or 6809 extended mnemonics, listed in table 2.3. The 6809 and 6800 Family mnemonics are identical to those defined by Motorola in the 6809 Programming and Macro Assemblers manual, publication M68MASR. The 6800 Family compatability mnemonics are provided to simplify the process of upgrading previously written 6800 software to run on the 6809. Note that not all 6800 Family operations have equivalent 6809 operations, and that the assembler will generate instruction sequences that emulate the 6800 operations. Several extended mnemonics are provided as an aid to structured programming, and to simplify syntax for several types of operations.

For primary statements, the mnemonic must be separated from the label (if any) by at least one separator character. No distinction is made between upper case and lower case mnemonics, so that the mnemonic for a 6809 no-operation can be either "NOP" or "nop", or for that matter, "Nop" or "nOp", etc. In general, each mnemonic code coresponds with a 6809 machine instruction. This assembler also recognises 6800 Family mnemonics, and performs a cross-assembly into functionally equivalent 6809 instructions.

The 6800 Family operations have been included to provide for a simple and rapid upgrade to 6809 from other members of the family. In most cases, the code can simply be reassembled for the 6809. There are certain functional differences that may create problems. In particular, the use of constructs like "BNE *+12" are likely to be troublesome. In any event, the cross assembled code is likely to be much less effecient, both in terms of time and code space, than code rewritten for the 6809.

2.6.4 - Operands

Source statement operand fields are required with many of the 6809 mnemonic operations. If present, the operand field must begin before column 30 of the input statement, or be separated from the mnemonic field by exactly one space. Operands can consist of register or flag designators, addressing mode indicators, and expressions. The exact format of an operand varies with the addressing mode capabilities of the particular instruction. A detailed functional description of the various 6809 addressing modes is covered in a later chapter of this publication. Any information in the operand field of instructions that do not have operands is treated as statement comments.

2.6.5 - Statement Comments

Statement comments follow the operand field (if present) and continue until the end of statement is encountered. The end of statement may be the carriage return at the end of a line, or it may be a semicolon character. A semicolon indicates that secondary statements will follow on the same source input line.

One complication caused by the ability to have multiple statements per line is that statement comments (as opposed to comment statements) may not contain the semicolon character unless certain restrictions are noted. First, the semicolon should appear after column 30 on the input line, and second, the semicolon should be followed by at least two spaces. This will inform the assembler that no more statements may be found on this line.

Another consideration that must be noted when using multiple statements per line is that it is possible to make the assembler think that the first statement comment on a line should be treated as a secondary statement. In order to avoid this difficulty, it is suggested that the last secondary statement on a line be terminated with a space instead of a semicolon. Some examples of this technique are shown:

```
FILL  STA 0,X+; DECB; BNE FILL      Fill a field
      ASRA; RORB                    Shift D-Register Right
```

2.7 - Assembler Directives

Assembler directives must always be encountered in the context of a primary statement, i.e., they must be the only operation appearing on a line of input text. Several directives have restrictions on the presence of label fields and will generate an error message if these restrictions are not met. Similarly, several directives have restrictions on the presence of operand fields. Assembler directives vary widely in function and are discussed in a separate chapter of this document. Valid assembler directives are listed in table 2.4.

- Table 2.1 -

6809 Assembler Mnemonic Operation Codes

ABX							Add B to IXR
ADCA	ADCB						Add with Carry
ADDA	ADDB	ADDD					Add without Carry
ANDA	ANDB	ANDCC					Logical And
ASL	ASLA	ASLB					Arithmetic Shift Left
ASR	ASRA	ASRB					Arithmetic SHift Right
BCC	LBCC						Branch on Carry Clear
BCS	LBCS						Branch on Carry Set
BEQ	LBEQ						Branch on Equal
BGE	LBGE						Branch on Greater or Equal
BGT	LBGT						Branch on Greater
BHI	LBHI						Branch on Higher
BHS	LBHS						Branch on Higher or Same
BITA	BITB						Bit Test
BLE	LBLE						Branch on Less or Equal
BLT	LBLT						Branch on Less
BMI	LBMI						Branch on Minus
BNE	LBNE						Branch on Not Equal
BPL	LBPL						Branch on Plus
BRA	LBRA						Branch
BRN	LBRN						Branch Never
BSR	LBSR						Branch to Subroutine
BVC	LBVC						Branch on Overflow Clear
BVS	LBVS						Branch on Overflow Set
BZC	LBZC						Branch on Zero Clear
BZS	LBZS						Branch on Zero Set
CLR	CLRA	CLRB	CLRD				Clear
CMPA	CMPB	CMPD	CMPS	CMPU	CMPX	CMPY	Compare
COM	COMA	COMB					Compliment
CWAI							Conditioned Wait
DAA							Decimal Adjust
DEC	DECA	DECB					Decrement
EORA	EORB						Exclusive Or
EXG							Exchange Registers
INC	INCA	INCB					Increment
JMP							Jump
JSR							Jump to Subroutine
LDA	LDB	LDD	LDS	LDU	LDX	LDY	Load Register
LEAS	LEAU	LEAX	LEAY				Load Effective Address
LSL	LSLA	LSLB					Logical Shift Left
LSR	LSRA	LSRB					Logical Shift Right
MUL							Multiply
NEG	NEGA	NEGB					Negate
NOP							No-Operation
ORA	ORB	ORCC					Inclusive Or
PSHS	PSHU						Push Registers
PULS	PULU						Pull Registers from Stack
ROL	ROLA	ROLB					Rotate Left
ROR	RORA	RORB					Rotate Right

- Table 2.1 cont' -

6809 Assembler Mnemonic Operation Codes

RTI							Return from Interrupt
RTS							Return from Subroutine
SBCA	SBCB						Subtract with borrow
SEX							Sign Extend
STA	STB	STD	STS	STU	STX	STY	Store
SUBA	SUBB	SUBD					Subtract without Carry
SWI	SWI2	SWI3					Software Interrupt
SYNC							Synchronize to Event
TFR							Transfer Registers
TST	TSTA	TSTB					Test

- Table 2.2 -

Supported 6800 Family Mnemonic Operation Codes

ABA			Add B to A
ASLD			Arithmetic Left Shift
CBA			Compare B to A
CLC			Clear Carry
CLI			Clear Interrupt Mask
CLV			Clear Overflow
CPX			Compare to Index
DES			Decrement Stack Pointer
DEX			Decrement Index Pointer
INS			Increment Stack Pointer
INX			Increment Index
LDAA	LDAB		Load Accumulator
LSRD			Logical Right Shift
ORAA	ORAB		Inclusive Or Accumulator
PSHA	PSHB	PSHX	Push on System Stack
PULA	PULB	PULX	Pull from System Stack
SEC			Set Carry
SEI			Set Interrupt Mask
SEV			Set Overflow
STAA	STAB		Store Accumulator
TAB	TBA		Transfer Accumulators
TAP	TPA		Transfer Condition Flags
TSX	TXS		Transfer Stack and Index
WAI			Wait for Interrupt

- Table 2.3 -

Extended Mnemonic Operation Codes

CCC	Clear Condition Codes
EWAI	Enable and Wait
EXIT	Exit from Procedure
MARK	Mark Stack for Procedure
RET	Return with Registers
SCC	Set Condition Codes

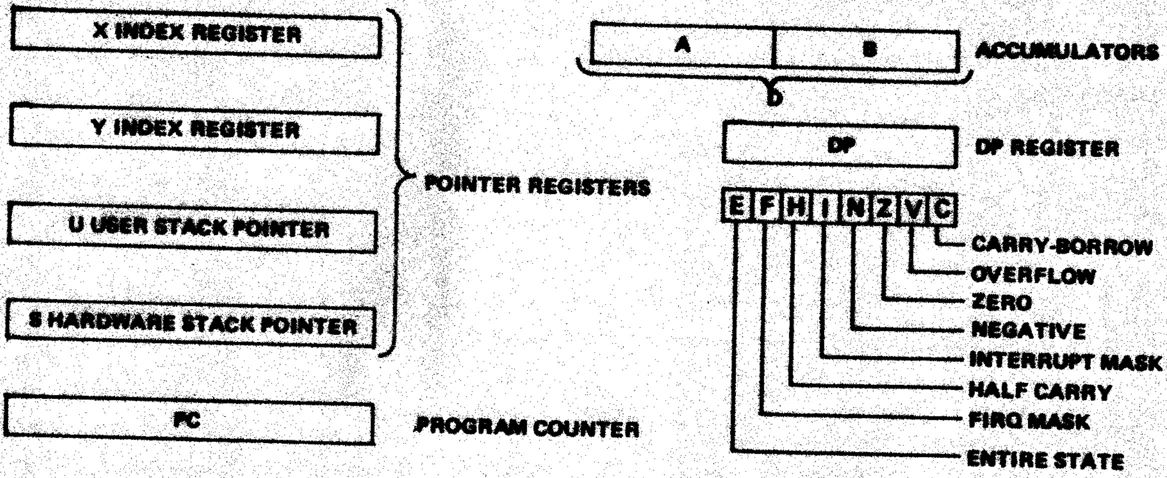
- Table 2.4 -

Assembler Directives

BSZ	Block Storage of Zeros
END	End of Segment
ENDF	End of File (Generated)
EQU	Equate Value
ERR	Generate Error Message
ERRIF	Conditional Error Message
FCB	Form Constant Bytes
FCC	Form Constant Characters
FDB	Form Double Bytes
FMB	Form Multiple Bytes
LIB	Include Library File
NAM	Name Module
OPT	Set Assembler Options
ORG	Begin Program Counter
PAG	Begin New Page
PROC	Begin Procedure
PUBLIC	Begin Public Library
QUAL	Begin Qualified Structure
RMB	Reserve Memory Bytes
SETDP	Set Direct Page Addressing
SPC	Space Listing
TTL	Provide Title
USE	Use Program Counter

3.0 - 6809 Software Architecture

The 6809 microprocessor is a stack-oriented, one-address microprocessor containing two accumulators, four pointer registers, a direct page register, and a condition flag register. With the addition of more pointer registers and a powerful complement of addressing modes, the 6809 is a major improvement over previous 6800 Family processors. Figure 3-1 is a programming model of the 6809 microprocessor. The following paragraphs give a brief description of each register and of how it is referenced by the programmer when writing assembler source code.



3.1 - Arithmetic Registers

The 6809 has two eight bit accumulators (called the A and B accumulators) that are used to perform arithmetic and logic operations. For many operations the A and B accumulators can be treated like a single sixteen bit accumulator (called the D accumulator), providing much improved performance in multiple-precision operations. The 6809 performs all arithmetic operations in two's complement format. The 6809 arithmetic registers are refered to by the single letters "A", "B", or "D".

3.2 - Pointer Registers

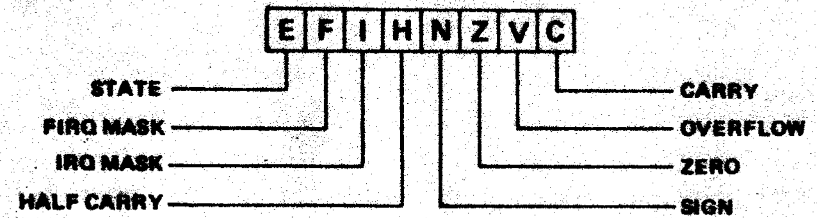
The 6809 has four sixteen bit pointer registers that can be used as base address registers for indexed mode addressing. There are two index registers, refered to as the "X" and "Y" registers, the user stack pointer refered to as the "U" register, and the system stack pointer, refered to as the "S" register. The various combinations available with indexed mode addressing allows all four pointer registers to be used as explicit stack pointers. In addition, the two stack registers have a series of PUSH and PULL instructions to facilitate zero address (stack) programing. The system stack pointer is implicitly used by the 6809 microprocessor for subroutine calls and interrupts.

3.3 - Program Counter

The 6809 maintains an internal sixteen bit program counter register refered to as the "PC" register. At any given time, the PC register may be thought of as a pointer to the next instruction to be executed. Two indexed addressing modes are available that utilize the program counter for their base address. These addressing modes provide the capability of writing program modules that are position independent.

3.4 - Condition Flag Register

The Condition Flag Register is conceptually and eight-bit register that contains the processor condition flags. It is refered to as the "CC" register. The bit positions of the condition register are shown in figure 3-2. A detailed description of each flag follows.



3.4.1 - Carry Flag

Bit zero is the carry flag refered to by the single letter "C". It represents the binary carry from an arithmetic or shift type operation. For these operations, the carry flag is an unsigned overflow indicator. In general, move-type and logical operations do not affect the carry flag.

3.4.2 - Two's Compliment Overflow Flag

Bit one is the two's complement overflow flag and is refered to by the single letter "V". It is set by an operation that causes a two's complement arithmetic overflow. Loads, stores, and logical operations generally clear the overflow flag, while arithmetic operations set it appropriately.

Since all 6809 arithmetic operations are of limited precision (eight or sixteen bits), it is possible to generate invalid signed results when performing arithmetic operations. For example, when performing an eight bit addition, it is possible to add 75 (base 10) (01001011 base 2) to 85 (base 10) (01010101 base 2) and get the invalid result -96 (base 10) (10100000 base 2). What has occured is that the carry out of the most significant bit (the sign bit) is different from the carry into the sign bit, hence the sign (and the value) of the result is invalid. It is under these conditions that the two's complement overflow flag is set. As another example, consider performing an arithmetic left shift on 96 (base 10) (01100000 base 2). The result is -64 (base 10) (11000000 base 2). Since the signed result is invalid, the overflow flag is set.

3.4.3 - Zero Flag

Bit two is the zero flag and is refered to by the single letter "Z". It is set whenever the result of an operation is zero. After compare operations, this bit represents the equal condition. After BIT type operations, this flag represents the state of the tested bits. Arithmetic, load, store, and logical operations set this flag appropriately.

3.4.4 - Sign Flag

Bit three is the sign flag and is refered to by the single letter "N" (for Negative). It is set whenever the most significant bit of the result is a one bit. For arithmetic operations, this flag is set if a valid negative two's complement result is obtained. Note that two's complement branches use both the N and V flags so that the the proper branch path is taken even if a two's complement overflow has occured.

3.4.5 - IRQ Interrupt Mask

Bit four is the IRQ mask bit and is refered to by the single letter "I". The processor will not recognize IRQ interrupts if this flag is set. The interrupt acknowledge sequence sets the IRQ mask flag to inhibit subsequent interrupt requests until the interrupt service routine completes or explicitly clears the interrupt mask. A return from interrupt instruction will restore the state of the interrupt mask flag from the stack.

3.4.6 - Half-Carry Flag

Bit five is the half-carry flag and is refered to by the single letter "H". This flag is used after eight bit add operations to indicate the carry out of bit three in the arithmetic unit. This flag is used by the DAA instructions to perform packed decimal (BCD) addition adjustment. In general, the half carry flag state is undefined after non-add operations and add type instructions on sixteen bit operands.

3.4.7 - FIRQ Interrupt Mask

Bit six is the FIRQ interrupt mask bit and is refered to by the single letter "F". This flag affects the FIRQ interrupt in the same manner that the I flag affects the IRQ interrupt. Remember that FIRQ interrupts do not stack the entire machine state.

3.4.8 - Entire State Flag

Bit seven is the Entire State flag and is refered to by the single letter "E". It is used only by the return from interrupt instruction to determine how much of the machine state was pushed onto the system stack at the time of an interrupt. Two saved states are defined: the entire state (E = 1) in which all registers have been pushed onto the system stack, and the subset state (E = 0) in which only the program counter and the condition flags have been pushed onto the stack. In general, the state of the E flag is indeterminate except after an interrupt.

3.5 - Direct Page Register

The Direct Page Register is an eight bit register that is used to provide the most significant eight bits of the sixteen bit address generated by instructions using direct addressing. It is refered to as the "DP" register and is initialized to zero at RESET time.

3.6 - Addressing Modes

One of the most useful features of the 6809 microprocessor is its wide variety of addressing modes. The use of these addressing modes permits the 6809 to be programmed either as a zero address (stack) machine, or as a one address (accumulator) machine. In addition to memory addressing modes, several implicit addressing modes reference internal processor registers and status indicators. Four instructions have been provided that explicitly perform stack operations that reference memory through the two stack pointer registers.

3.6.1 - Inherent addressing

Inherent addressing includes those instructions which have no user specifiable addressing options. All data references are implicit within the instruction itself.

```
     Example:     MUL                         Multiply Accumulators
                  SWI2                        Do User Software Interrupt
```

3.6.2 - Accumulator Addressing

Accumulator addressing refers to data values contained within the accumulator registers and does not generate a memory reference cycle. Most instructions perform operations on the eight bit A or B accumulators, while some instructions also perform operations on the sixteen bit D accumulator. The accumulator specification is normally appended to the mnemonic root specification.

```
     Example:     CLRA                        Clear A Accumulator
                  NEGB                        Negate B Accumulator
```

3.6.3 - Register Addressing

Register addressing refers to data values contained within one of the MPU data or pointer registers. The selected register or registers must be explicitly specified as instruction operands. A register list consists of a series of register specifications, separated by commas. Some instructions having register addressing implicitly reference memory through the two stack pointer registers.

```
     Example:     TFR     D,X                 Move Data from D to IX
                  PSHS    A,B,X               Push Registers on Stack
```

3.6.4 - Condition Flag Addressing

Condition flag addressing refers to specific flag bits in the condition flag register. This form of addressing is used for the condition code operations. A condition flag list consists of a series of condition flag specifications, separated by commas.

```
Example:    EWAI   I,F          Wait for IRQ or FIRQ
            SCC    V            Set the Overflow Flag
```

3.6.5 - Memory Addressing Modes

Memory addressing modes are used to specify operations on operands residing in main memory. Several memory addressing modes are available. Immediate addressing accesses an operand that is contained within the instruction itself. Absolute addressing requires an operand whose exact memory address is known at assembly time. Indexed addressing accesses an operand at an address that is developed from the contents of one of the MPU pointer registers and thus is the most flexible of the addressing modes.

3.6.6 - Immediate Addressing

Immediate addressing refers to a data value that is contained within the byte or bytes immediately following the instruction opcode. This mode is used to access a value that is known at assembly time and which will not be changed during program execution. Immediate addressing is specified by prefixing the operand expression with a crosshatch, "#".

```
Example:    LDA    #12          Make A = 12
            CMPD   #ADDRESS     See if D = ADDRESS
```

3.6.7 - Absolute Addressing

Absolute addressing refers to a data value that is referenced by an address word or byte immediately following the instruction opcode. There are two program selectable modes of absolute addressing: Direct and Extended. Both of these modes are necessarily position dependent.

Direct addressing uses the eight bit immediate value of the instruction as the low order eight bits of an address. The high order eight bits are obtained from the direct page register (DPR). In this way, an instruction utilizing direct addressing can reference one of 256 locations in a "page" of memory selected by the direct page register. Extended addressing uses the sixteen bit immediate value of the instruction as the address of the data value and can access data anywhere in memory.

In order to specify absolute addressing, specify the address of the data as the operand field of the instruction. The assembler computes the specified address and compares the high order portion with the assumed contents of the direct page register (specified via the SETDP directive) in order to determine absolute addressing mode. If the programmer wishes to explicitly specify direct or extended addressing, two significance forcing characters are provided. The less-than sign "<" forces the assembler to create an eight bit address while the greater-than sign ">" forces a sixteen bit address. In the case of an eight bit address, a warning is issued if the assembler determines that eight bits is insufficient.

```
Example:    LDB    BYTE        Load a Byte
            TST    <LOWBYTE    Test a Byte - Direct
            CLR    >HIBYTE     Clear a Byte - Extended
```

3.6.8 - Relative Addressing

Relative addressing is used for branch address calculations and refers to an address that is computed from the updated program counter value and the byte or word of offset contained within the instruction. Short relative addressing uses an eight bit offset and provides relative addresses of -128 to +127 bytes. Long relative addressing uses a sixteen bit offset and can address anywhere in memory.

The assembler normally computes the offset required and assignes either long or short relative addressing as appropriate. If the programmer wishes to explicitly assign short or long relative addressing, the two significance forcing characters may be used similar to absolute addressing above. A warning message is produced if short addressing is selected and an eight bit offset is insufficient.

```
Example:    BRA    LABEL       Relative Addressing
            BNE    <SHORT      Short Relative Addressing
            BEQ    >LONG       Long Relative Addressing
```

3.6.9 - Indexed Addressing

Indexed addressing refers to data values whose address is developed from the value contained in one of the MPU pointer registers. The specific register used to develop the address of the actual data (called the effective address) is called the index base register. The register to be used for a base address must always be explicitly specified. Certain indexing modes have the ability to use the program counter register as their index base register.

Indexed addressing requires the presence of an indexing mode post byte following the instruction opcode. This post byte specifies both the type of indexed addressing to be used and which index base register to use. If an offset or address is required by the indexing mode, this value follows the post byte in the immediate data field. Several options are available to conserve both execution time and object code space. The assembler automatically selects the instruction format that will require minimum object code space and time.

3.6.10 - Constant Offset Indexing

Constant offset indexed addressing generates an effective address by adding a fixed offset to the contents of one of the four MPU pointer registers. The offset is contained within the instruction itself and follows the indexing mode post byte. Offsets are signed values, and may be five, eight, or sixteen bits in length. The assembler computes the offset and selects the smallest adequate format. The base register is specified following the offset expression. If an explicit offset size is desired, the significance forcing characters "<" and ">" " may be used to select eight or sixteen bit offsets respectively.

```
Example:     LDA    12,X          Constant Offset from X
             STA    -2,Y          Constant Offset from Y
             LDX    <0,U          Forced 8-bit Offset
             STX    >12,S         Forced 16-bit Offset
```

3.6.11 - Constant Offset Indirect Indexing

Like most of the indexed addressing modes, constant offset indexing may specify a single level of indirection. The effective address is generated by adding the fixed offset to the value of the index base register, and then using that address to fetch a sixteen bit effective address from memory. Indirection is specified by enclosing the operand in square brackets. Any significance forcing characters must precede the expression and be inside of the brackets.

```
Example:     JMP    [0,X]         Constant Offset Indirect
             LDD    [>12,U]       Forced 16-bit Offset
```

3.6.12 - Accumulator Offset Indexing

Accumulator offset indexed addressing adds the contents of an accumulator register to the value of an index base register to generate an effective address. If indirection is specified, this address is then used to fetch the effective address value from memory. In the case of the A or B accumulators the offset is a signed eight bit value. For the D accumulator, the offset is a signed sixteen bit value. Accumulator offsets are selected by specifing the accumulator register as the operand followed by the index base register specification. Like constant offset indexing, indirection is specified by placing the operand inside of square brackets.

```
Example:     STA    B,Y           8-bit Accumulator Offset
             LDX    D,U           16-bit Accumulator Offset
             CMPB   [A,X]         8-bit Offset Indirect
```

3.6.13 - Autoincrement Indexing

Autoincrement indexed addressing uses the value of an index base register as the effective address. If indirection is specified, this address is then used to fetch the effective address value from memory. After the effective address is determined, the base register is incremented by one or two. Note that the increment must be two if indirection is specified. No offset is permitted when using autoincrement addressing. Autoincrement is selected by following the base register specification by either one or two plus signs "+", for increments of one or two respectively. Like other forms of addressing, indirection is specified by enclosing the operand in square brackets.

```
     Example:      LDA     0,X+           AutoIncrement by One
                   STD     0,Y++          AutoIncrement by Two
                   LDU     [0,S++]        AutoIncrement Indirect
```

3.6.14 - Autodecrement Indexing

Autodecrement indexed addressing subtracts either one or two from an index base register and subsequently uses that value as the effective address. If indirection is specified, this address is then used to fetch the effective address value from memory. Note that the decrement value must be two if indirection is specified. No offset is permitted with autodecrement indexing. Autodecrement addressing is selected by preceding the base register specification by either one or two minus signs "-", for decrements of one or two respectively. Like other forms of indexed addressing, indirection is specified by enclosing the operand in square brackets.

```
     Example:      CLR     0,-X           AutoDecrement by One
                   LDY     0,--U          AutoDecrement by Two
                   STY     [0,--S]        AutoDecrement Indirect
```

3.6.15 - Extended Absolute Indirect Addressing

Extended absolute indirect addressing uses the address word contained in the instruction to fetch an effective address from memory. This addressing mode allows the programmer to define a pseudo register vector (in IBM terminology) for use in communicating between program modules. Since the instruction contains an absolute address, it is necessarily position dependent.

```
     Example:      BITA    [DEVICE]       Extended Absolute Indirect
```

3.6.16 - Program Counter Relative Addressing

Program counter addressing uses the value of the updated program counter register as the index base value. A fixed offset contained within the instruction is added to the updated program counter value to obtain the effective address. If indirection is specified, this address is then used to fetch the effective address from memory. The expression value specified in the source code is the desired value of the effective address; the assembler uses that value to compute the required offset. Program counter relative addressing is specified by affixing the "PC" register specification to the requested address. Like other forms of indexing, indirection is specified by enclosing the operand in square brackets.

```
     Example:      LDA     BYTE,PC        Program Counter Relative
                   STX     [ADDR,PC]      Program Counter Indirect
```

4.0 - Assembler Expressions

Expressions consist of one or more terms combined with assembler operators. Each term represents a sixteen bit signed value, and the result of expression evaluation is also sixteen bits and signed. The expression value may be absolute, relocatable, or complex relocatable, depending on the relocation attributes of the various terms and the operators used upon them. In addition to the relocation attributes, the expression may have the starred attribute. This attribute will be set if any of the terms in the expression have the star attribute. More information on starred expressions can be found in the chapter on assembler directives.

Under certain circumstances, expressions may be preceded or surrounded by special characters used to specify addressing modes. It must be clearly understood that these mode characters are not part of the expression proper, and hence must not appear within an expression.

4.1 - Terms in Expressions

Terms in expressions may consist of symbolic references, location counter references, numeric constants, character constants, or truth value constants. Symbolic references have an explicit relocation attribute set when the symbol is defined. Location counter references have the relocation attributes of the current program counter. Constants always have a relocation attribute of absolute.

4.1.1 - Symbolic References

Symbolic references may consist of a local reference, a global reference, a parental reference, or a structure reference. Local references consist of an identifier with no qualifier characters (".") and refer to the most local definition of that identifier. Global references consist of the global qualifier character (".") followed by an identifier and refer to that identifier defined in the global dictionary. Parental references consist of the parental qualifier character ("^") followed by an identifier and refer to that identifier defined in the parent dictionary of the current procedure. Structure references, which may be local, global, or parental, consist of identifiers separated by global qualifier characters. Some examples of symbolic references are:

```
     LABEL  - - - - - - - - - - - - - - - - -  a local reference
    .LABEL  - - - - - - - - - - - - - - - - -  a global reference
    ^LABEL  - - - - - - - - - - - - - - - - -  a parental reference
     TABLE.ENTRY  - - - - - - - - - - - - - -  a local structure
    .TABLE.ENTRY  - - - - - - - - - - - - - -  a global structure
    ^TABLE.ENTRY  - - - - - - - - - - - - - -  a parental structure
     PAGE.PARAGRAPH.PHRASE.WORD.LETTER  - -  a local structure
```

4..2 - Location Counter References

Location-counter references consist of the asterisk "*" used in place of a symbolic reference. The value and relocation attributes of a location counter reference are those of the current program counter at the beginning of primary statement processing for the current line of assembler source input. Note that this value does not change within a line of source code, regardless of changes in program counter values. For example, in the following statements, both location counter references have the same value (which is the value of the identifier "LABEL"):

```
          ORG   $0200
    LABEL CLR   0,X+; DECB; BPL *; DECA; BPL *;
```

4.1.3 - Numeric Constants

Numeric type constants consist of an optional radix designator character, followed by a string of digits. If the radix is greater than ten, the larger digits are specified as letters, with the letter "A" having a value of ten, "B" for eleven, and so forth. Each digit is checked to be sure that its value is less than that of the designated radix. For the purposes of numeric constant evaluation, the assembler treats letters of lower case and upper case as identical.

Permissible radix designator characters are "$" denoting hexidecimal numbers, "%" denoting binary numbers, and "@", denoting octal numbers. In the absence of a radix designator, decimal numbers are assumed. Numeric type constants always have a relocation attribute of absolute. Examples of numeric constants are as follows:

Decimal Constant	--	21845
Hexidecimal Constant	--	$5555
Binary Constant	--	%101010101010101
Octal Constant	--	@52525

4.1.4 - Character Constants

Character type constants consist of an opening quote character followed by a string of characters followed by the closing quote character. This assembler recognises three characters as quote characters: the double quote """, the single quote (apostrophe) "'", and the grave accent "`". Any of these characters may be used to begin a character constant, however, the closing quote must be the same character as the opening quote. Character constants are limited to a precision of 16 bits or two characters, and always have a relocation attribute of absolute. Examples of character constants are:

''	--	Character constant with value $0000
"A"	--	Character constant with value $0041
'AB'	--	Character constant with value $4142
`ABC`	--	Character constant with value $4243

Quotes within character constants may be denoted by using two successive quote characters, or by using a different quote character as a delimiter. In either case, there must always be a proper character as a closing quote. Some examples of character constants containing quotes are:

```
"'"      --  Character constant with value $0027
''''     --  Character constant with value $0027
"''"     --  Character constant with value $2727
''''''   --  Character constant with value $2727
```

If the Motorola compatability option has been selected, character constants consist of an opening single quote character followed by exactly one ASCII character. In this case, the upper nine bits of the character constant value are zero, and the lower seven bits have the ASCII value of the following character. No closing quotes are permitted. Some examples of Motorola compatable character constants are:

```
'A       --  Character constant with value $0041
'        --  Character constant with value $0020
''       --  Character constant with value $0027
```

4.1.5 - Truth Constants

Truth value constants consist of the reserved symbols TRUE and FALSE and have values of 1 and 0 respectively. The relocation attribute of truth value constants is always absolute. In expressions, truth value operators treat any non-zero value as being equivalent to the truth value TRUE.

4.2 - Operators in Assembler Expressions

The assembler supports a wide variety of operators in expressions. These operators are used to evaluate expressions at assembly time, and in addition, several operators can be passed to the linker program to cause expression evaluation at link time. In either case, operators are processed in precedence order. That is, operators with a higher precedence value are processed before operators with a lower value. A complete listing of operators and precedence values can be found in table 4.1. In the following example, the value of VAR2 is multiplied by the value of VAR3, and the result is added to VAR1:

```
VAR1+VAR2*VAR3     multiply has higher precedence than add
```

Each term of the source expression is evaluated to a signed, sixteen bit binary relocatable value during expression scanning. These binary values are then passed to the expression evaluator to perform the required arithmetic. Note that expression evaluation always produces a sixteen bit result even if terms in the expression are undefined. Undefined terms have a value of zero and a relocation attribute of absolute.

4.3 - Grouping Operators

The two parentheses and are used as grouping operators are used to alter the order of expression evaluation by explicitly stating the order in which expressions are to be processed. These operators have a precedence value of 12; higher than that of any other operator. Parentheses may be nested up to ten levels deep. The following example demonstrates the use of parentheses:

```
0007    E1    EQU    1+2*3        NOT GROUPED
0006    E2    EQU    (1+2)*3        GROUPED
```

4.4 - Arithmetic Operators

Seven arithmetic operators are provided. The unary negation operator, "-", returns as its result the two's complement of its operand. If an overflow occurs as a result of the negation, the maximum negative sixteen bit number is returned as the result. The unary plus operator, "+", is essentially a no-op. The binary addition operator, "+", and the binary subtraction operator, "-", perform sixteen bit two's complement arithmetic. Any overflow that may occur produces a warning message but is otherwise ignored. The multiplication operator, "*", performs a sixteen bit signed multiply. If the results of the multiplication have more than sixteen bits of significance, a warning message is produced and the result is then truncated to sixteen bits. The division operator, "/", performs a sixteen bit signed division with the sign of the result determined by the rules of algebra. Any remainder is discarded. Note that a divide by zero produces a warning and substitutes the maximum possible sixteen bit number for the result. The modulus operator, "%", performs a signed division and returns the remainder as the result. The sign of the remainder is always the same as the sign of the dividend. If a divide by zero occurs, the result is set to zero. A few examples will illustrate the use of the seven arithmetic operators:

```
0003    E1    EQU    +3          UNARY PLUS
FFF8    E2    EQU    -7          UNARY MINUS
000F    E3    EQU    5+10        ADDITION
FFF6    E4    EQU    5-10        SUBTRACTION
0032    E5    EQU    5*10        MULTIPLICATION
0005    E6    EQU    100/17      DIVISION
000F    E7    EQU    100%17      MODULUS
```

4.5 - Truth Value Operators

The two unary truth value operators are used to convert arithmetic (possibly relocatable) values into truth values. The truth value operator "/" has the value TRUE if its argument is non-zero, and FALSE if its argument is zero. The truth value negation operator "!" performs the truth value conversion in the same manner, and then inverts the result.

The value TRUE is equal to one, with a relocation attribute of absolute. Similarly, FALSE is equal to zero, also absolute. These values may be used in arithmetic expressions. Examples of the truth value operators are:

```
0001    V1    EQU   /12         TRUE VALUE
0000    V2    EQU   !12         FALSE VALUE
```

4.6 - Relational Operators

The six binary relational operators are used to compare their left and right operands. If the relational condition is satisfied, the value of the expression is TRUE, otherwise it is FALSE. These operators are particularly useful in conditional assembly and for use with the ERRIF directive. Examples of the six relational operators are:

```
0020    HI    EQU   32
0010    LO    EQU   16

0000    R1    EQU   HI<LO       LESS THAN
0000    R2    EQU   HI<=LO      LESS THAN OR EQUAL TO
0000    R3    EQU   HI==LO      EQUAL TO
0001    R4    EQU   HI!=LO      NOT EQUAL TO
0001    R5    EQU   HI=>LO      GREATER THAN OR EQUAL TO
0001    R6    EQU   HI>LO       GREATER THAN
```

In order to simplify the coding of relational operators, several variations on basic syntax are recognised as valid. The Less Than or Equal To operator may be specified as "<=" or also as "=<", the Greater Than or Equal To operator as "=>" and also as ">=", the Not Equal To operator as "!=", "<>", and also as "><". The function of these composite operators is exactly the same; only their syntax is different.

4.7 - Bitwise Logical Operators

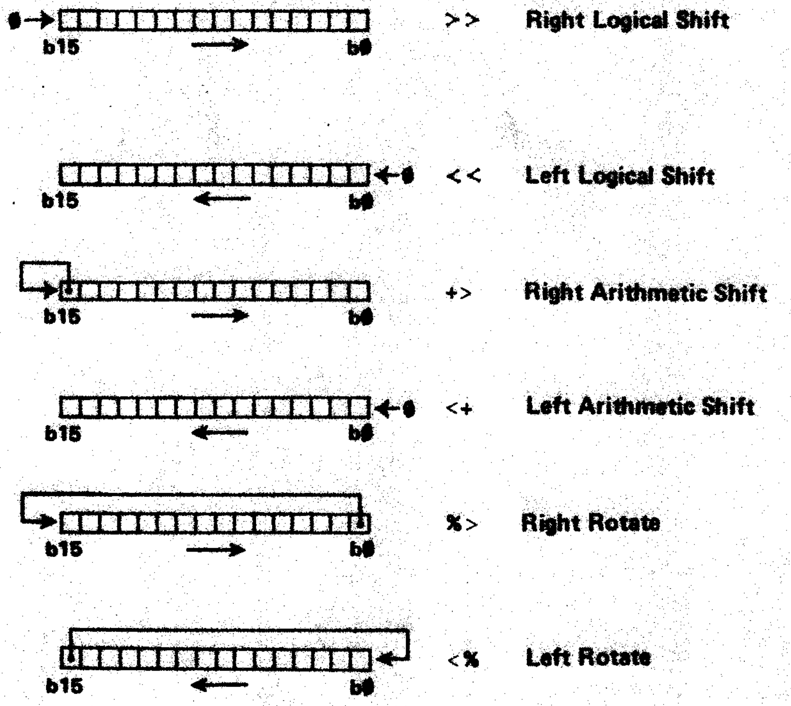
Four bitwise logical operators are provided. The unary NOT operator "~" produces as its result the one's complement of its operand. The three binary operators are the Inclusive OR operator "|", the Exclusive OR operator, "^", and the AND operator, "&". These operators perform their respective operatons bitwise upon their two operands. The following examples will clarify their functions:

```
0F0F    M1    EQU   $0F0F
00FF    M2    EQU   $00FF

F0F0    V1    EQU   ~M1         UNARY NOT
0FFF    V2    EQU   M1|M2       BINARY INCLUSIVE OR
0FF0    V3    EQU   M1^M2       BINARY EXCLUSIVE OR
000F    V4    EQU   M1&M2       BINARY AND
```

4.8 - Shift Operators

All of the six shift operators are binary with the left operand specifying the value to be shifted and the right operand specifying the bit count. The shift count is signed, meaning that a left shift with a negative bit count is converted into a right shift and so forth. No checks are made for arithmetic overflow or lost significance. The following diagrams illustrate the six different shifts:



Logical shifts supply zero bits for all positions vacated. Arithmetic shifts sign extend the value being shifted. Rotates use the bits shifted out of the value to fill the vacated bit positions. The following examples illustrate the shift operators:

```
F0F0    M1    EQU    $F0F0

7878    V1    EQU    M1>>1          LOGICAL RIGHT SHIFT
C3C0    V2    EQU    M1<<2          LOGICAL LEFT SHIFT
FE1E    V3    EQU    M1+>3          ARITHMETIC RIGHT SHIFT
0F00    V4    EQU    M1<+4          ARITHMETIC LEFT SHIFT
8787    V5    EQU    M1%>5          RIGHT ROTATE
3C3C    V6    EQU    M1<%6          LEFT ROTATE
```

4.9 - Logical Connectives

Logical connectives are binary operators used to join truth valued expressions into complex truth values. Two connectives are available, Logical AND, "&&", and Logical OR, "||". Some examples of the use of connectives are:

```
0010    V1    EQU    16
0020    V2    EQU    32
0020    V3    EQU    V1<<1

0000    LX    EQU    V1==V2&&V2==V3    LOGICAL AND
0001    LY    EQU    V1==V2||V2==V3    LOGICAL OR
```

- Table 4.1 -

Assembler Operator Precedence

Operator		Function	Precedence
(	-	Left Parenthesis	12
)	-	Right Parenthesis	12
+	-	Unary Plus	11
-	-	Unary Negation	11
~	-	Unary Bitwise Not	11
/	-	Unary Truth Value	11
!	-	Unary Negated Truth Value	11
+	-	Arithmetic Addition	9
-	-	Arithmetic Subtraction	9
*	-	Arithmetic Multiplication	10
/	-	Arithmetic Division	10
%	-	Arithmetic Modulus (Remainder)	10
<<	-	Left Logical Shift	8
<+	-	Left Arithmetic Shift	8
<%	-	Left Rotate	8
>>	-	Right Logical Shift	8
+>	-	Right Arithmetic Shift	8
%>	-	Right Rotate	8
<	-	Relational Less Than	7
<=	-	Relational Less Than or Equal To	7
=<		(Same)	
==	-	Relational Equal To	6
=>	-	Relational Greater Than or Equal To	7
>=		(Same)	
>	-	Relational Greater Than	7
!=	-	Relational Not Equal To	6
<>		(Same)	
><		(Same)	
&	-	Bitwise AND	5
^	-	Bitwise Exclusive OR	4
|	-	Bitwise Inclusive OR	3
&&	-	Logical Connective AND	2
||	-	Logical Connective OR	1

5.0 - 6809 Operation Mnemonics

The following pages contain a detailed description of the 6809 operations supported by the SWTPC assembler. These operations consist of 6809 primitives and a few extended mnemonics designed to simplify structured programming practices. Each table entry consists of the assembler mnemonic, a description of the function of each of the operations, a list of affected condition flags, and the valid addressing modes for that operation.

5.1 - Condition Flags

The condition flag list contains information about which condition flags an operation alters and the criteria for the result. Unless otherwise specifically mentioned, the Interrupt Mask (IRQ) and Fast Interrupt Mask (FIRQ) are unchanged by the operation. The Entire State flag (E) is undefined in the condition flag register and is valid on the stack only after an interrupt. The slow maskable interrupt (IRQ), the non-maskable interrupt (NMI), and all three of the software interrupts (SWI) set the Entire State flag before pushing the MPU registers on the stack. Only the fast maskable interrupt (FIRQ) clears the entire state flag and then pushes only the condition flags and the program counter on the stack.

5.2 - Extended Mnemonics

Several of the supported mnemonics are not strictly speaking 6809 operations, and consist of either multiple 6809 instructions or of syntax different from the Motorola standard. These instructions are marked with the notation "ext" after the assembler mnemonic. They have been provided for the sake of program clarity and coding convenience. All extended mnemonics will produce error messages if the Motorola compatability option has been set.

5.3 - 6800 Family Mnemonics

Full support has been provided for 6800 Family mnemonic operation codes (except of course the 6805). In certain cases, these operations will assemble into multiple instruction sequences designed to emulate the specified 6800 operations. Emulation is exact in all cases except for the 6801 MUL instruction which is upward compatable. The 6809 MUL operation sets the zero flag when appropriate while the 6801 operation does not.

ABX	Add ACCB Into IX
DESCRIPTION:	Add the eight bit unsigned value in the B accumulator into the X index register. This instruction is provided for 6801 compatibility.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Inherent

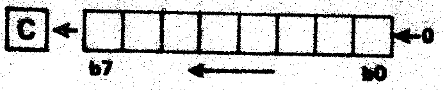
ADC	Add With Carry
DESCRIPTION:	Adds the carry flag and the memory byte into an eight bit register.
CONDITION CODES:	H: Set if the operation causes a carry from bit three in the ALU. N: Set if the bit seven of the result is Set. Z: Set if all bits of the result are Clear. V: Set if the operation causes a two's complement arithmetic overflow. C: Set if the operation causes a carry from the high order bit in the ALU.
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

ADD	Add Without Carry
DESCRIPTION:	Adds memory into register.
CONDITION CODES:	H: For eight bit operations, set if the operation causes. a carry from bit three in the ALU. For sixteen bit operations, the H flag is unaffected. N: Set if the high order bit of the result is Set. Z: Set if all bits of the result are Clear. V: Set if the operation causes a two's complement arithmetic overflow. C: Set if the operation causes a carry from the high order bit in the ALU.
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

AND	Logical AND
DESCRIPTION:	Performs an eight bit logical AND operation between the contents of a register and the contents of memory.
CONDITION CODES:	H: Not Affected. N: Set if bit seven of the result if Set. Z: Set if all bits of the result are Clear. V: Cleared. C: Not Affected.
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

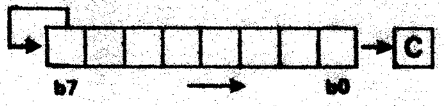
ANDCC	Logical AND Into Condition Code Register
DESCRIPTION	Performs an eight bit logical AND between the condition code register and the immediate byte and places the result in the condition code register.
CONDITION CODES:	The condition codes are set to the result of the 8-bit logical AND of the current condition code bits with the immediate operand. Any condition code bit including the interrupt masks may be cleared by this operation.
ADDRESSING MODES:	Immediate

ASL	Arithmetic Shift Left
DESCRIPTION:	Shifts all bits of the operand one place to the left. Bit zero is loaded with a zero. The high order bit of the operand is shifted into the carry flag.



CONDITION CODES:	H: Undefined. N: Set if the high order bit of the result is Set. Z: Set if all bits of the result are Clear. V: Set if the bit shifted out of the high order bit is not equal to the bit shifted into the high order bit. C: Loaded with the high order bit of the original operand.
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

ASR	Arithmetic Shift Right
DESCRIPTION:	Shifts all bits of the operand one place right and sets the carry flag from bit zero of the original operand. The high order bit is held constant to provide proper two's complement sign extension.



CONDITION CODES:	H: Undefined. N: Set if the sign bit of the result is Set. Z: Set if all bits of result are Clear. V: Not Affected. C: Loaded with bit zero of the original operand.
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

BCC	Branch on Carry Clear
DESCRIPTION:	Tests the state of the Carry bit and causes a branch if Carry is clear.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BCS	Branch on Carry Set
DESCRIPTION:	Tests the state of the Carry bit and causes a branch if Carry is set.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BEQ	Branch on Equal
DESCRIPTION:	Used after a subtract or compare operation, this instruction will branch if the register is equal to the memory operand.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BGE	Branch on Greater or Equal
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instrucion will branch if the register was greater than or equal to the memory operand.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BGT	Branch on Greater
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was greater than the memory operand.
CONDITION CODE:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BHI	Branch if Higher
DESCRIPTION:	Used after a subtract or compare operation on unsigned binary values this instrucion will branch if the register was higher than the memory operand.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BHS	Branch if Higher or Same
DESCRIPTION:	When used after a subtract or compare on unsigned binary values, this instruction will branch if register was higher than or same as the memory operand.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BIT	Bit Test
DESCRIPTION:	Performs an eight bit logical AND of the contents of a register and a memory operand and modifies condition codes accordingly. The contents of the register are not affected.
CONDITION CODES:	H: Not Affected. N: Set if bit seven of the result is Set. Z: Set if all bits of the result are Clear. V: Cleared. C: Not Affected.
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

BLE	Branch on Less or Equal
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was less than or equal to the memory operand.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BLO	Branch on Lower
DESCRIPTION:	When used after a subtract or compare on unsigned binary values, this instruction will branch if the register was lower than the memory operand.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BLS	Branch on Lower or Same
DESCRIPTION:	Used after a subtract or compare operation on unsigned binary values, this instruciton will branch if the register was lower than or the same as the memory operand.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BLT	Branch on Less
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was less than the memory operand.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BMI	Branch on Minus
DESCRIPTION:	Used after an operation on signed binary values, this instruciton will branch if the result is negative.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BNE	Branch Not Equal
DESCRIPTION:	Used after a subtract or compare operation, this instruction will branch if the register is not equal to the memory operand.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BPL	Branch on Plus
DESCRIPTION:	Used after an operation signed binary values, this instruction will branch if the result is positive.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BRA	Branch
DESCRIPTION:	Causes an unconditional branch.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BRN	Branch Never
DESCRIPTION:	Does not cause a branch. This instruction is essentially a NO-OP.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BSR	Branch to Subroutine
DESCRIPTION:	The updated program counter is pushed onto the system stack and control is transferred to the effective address.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BVC	Branch on Overflow Clear
DESCRIPTION:	Tests the state of the overflow flag and causes a branch if the overflow flag is set.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BVS	Branch on Overflow Set
DESCRIPTION:	Tests the state of the overflow flag and causes a branch if the overflow flag is clear.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BZC	Branch on Zero Clear
DESCRIPTION:	Tests the state of the zero flag and causes a branch if the zero flag is clear.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

BZS	Branch on Zero Set
DESCRIPTION:	Tests the state of the zero flag and causes a branch if the zero flag is set.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative

CCC (ext)	Clear Condition Code
DESCRIPTION:	Explicitly clears any subset of the MPU condition flags. This operation is an extended syntax version of ANDCC.
CONDITION CODES:	All condition flags specified as operands are cleared. It is not possible to specify the Entire State flag.
ADDRESSING MODES:	Condition List

CLR	Clear
DESCRIPTION:	The register or memory is loaded with zero. The carry flag is cleared for 6800 compatibility.
CONDITION CODES:	H: Not Affected. N: Cleared Z: Set V: Cleared C: Cleared
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

CMP	Compare Memory to a Register
DESCRIPTION:	Compares a memory operand to the contents of a specified register and sets appropriate condition codes.
CONDITION CODES:	H: Undefined for eight bit operations, and unaffected for 16-bit operations. N: Set if the high order bit of the result is Set. Z: Set if all bits of the result are Clear. V: Set if the operation causes a two's complement overflow. C: Set if the subtraction did not cause a carry from the most significant bit of the ALU.
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

COM	Complement
DESCRIPTION:	Replaces the contents of a register or memory with its one's complement. The carry flag is set for 6800 compatibility.
CONDITION CODES:	H: Not Affected. N: Set if bit seven of the result is Set Z: Set if all bits of the result are Clear V: Cleared. C: Set.
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

CWAI	Clear and Wait for Interrupt
DESCRIPTION:	The CWAI instruction ANDs an immediate byte with the condition code register (which may clear interrupt masks), stacks the entire machine state on the system stack, and then waits for an interrupt. When a non-masked interrupt occurs, no further machine states will be saved before vectoring to the interrupt handling routine.
CONDITION CODES:	The condition codes are set to the result of the eight bit logical AND of the current condition code bits and the immediate operand. Any condition code bits including interrupt masks may be cleared by this operation.
ADDRESSING MODES:	Immediate

DAA	Decimal Addition Adjust
DESCRIPTION:	This instruction is used after the addition of two binary-coded decimal numbers to insure that the result is in the proper binary-coded decimal format, and that the carry flag is set correctly. This instruction should be used after an ADD or ADC instruction, with the result held in the A register.
CONDITION CODES:	H: Not Affected. N: Set if bit seven of result is Set. Z: Set if all bits of the result are Clear. V: Undefined. C: Set if the operation causes a carry from bit seven in the ALU, or if the carry flag was set prior to the operation.
ADDRESSING MODES:	Inherent

DEC	Decrement
DESCRIPTION:	Subtract one from the operand. The carry flag is not affected, thus allowing DEC to be a loop-counter in multiple precision computations.
CONDITION CODES:	H: Not Affected. N: Set if bit seven of result is Set. Z: Set if all bits of result are Clear. V: Set if a two's complement arithmetic overflow occurs. C: Not Affected.
ADDRESSING MODES:	Accumulator Direct, Indexed, Extended

EOR	Exclusive OR
DESCRIPTION:	A memory operand is exclusive ORed into an eight bit register.
CONDITION CODES:	H: Not Affected. N: Set if bit seven of result is Set Z: Set if all bits of result are Clear V: Cleared. C: Not Affected.
ADDRESSING MODES:	Immediate, Direct, Extended, Indexed

EWAI (ext)	Enable Interrupts and Wait
DESCRIPTION:	Explicitly clears any subset of the MPU condition flags, stacks the contents of the MPU registers on the system stack, and waits for an interrupt. This operation is an extended syntax version of CWAI.
CONDITION CODES:	All condition flags (including interrupt masks) specified as operands are cleared. It is not possible to specify the Entire State flag.
ADDRESSING MODES:	Condition List

EXG	Exchange Registers
DESCRIPTION:	Exchange two register values. Note that registers may only be exchanged with registers of like size, i.e., eight bit with eight bit, or sixteen bit with sixteen bit.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Register

EXIT (ext)	Exit from Procedure
DESCRIPTION:	The Exit instruction loads the system stack pointer from the user stack pointer, and then pulls the previous user stack pointer, the specified registers, and the program counter (which effects a return from subroutine) from the system stack.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Register List

INC	Increment
DESCRIPTION:	Add one to the operand. The carry flag is not affected, thus allowing INC to be used as a loop-counter in multiple precision computations.
CONDITION CODE:	H: Not Affected. N: Set if bit seven of the result is Set. Z: Set if all bits of the result are Clear. V: Set if a two's complement arithmetic overflow occurs. C: Not Affected.
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

JMP	Jump
DESCRIPTION:	Program control is transferred to the effective address.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Direct, Indexed, Extended

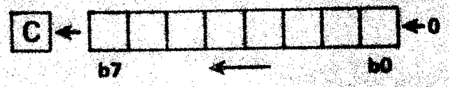
JSR	Jump to Subroutine
DESCRIPTION:	The updated program counter is pushed onto the system stack and control is transferred to the effective address.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Direct, Indexed, Extended

LD	Load Register from Memory
DESCRIPTION:	Load the contents of the addressed memory into the register
CONDITION CODES:	H: Not Affected. N: Set if bit seven of loaded data is Set Z: Set if all bits of loaded data are Clear V: Cleared. C: Not Affected.
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

LEA	Load Effective Address
DESCRIPTION	Form the effective address to data using the memory addressing mode. Load that address, not the data itself into the pointer register.
CONDITION CODES:	LEAX and LEAY affect the Zero flag to allow use as counters and for 6800 INX/DEX compatibility. LEAU and LEAS do not affect the Zero flag to allow for cleaning up to the stack while returning the Zero flag as a parameter to a calling routine, and for 6800 INS/DES compatibility. All other condition flags are unaffected.
ADDRESSING MODES:	Indexed

LSL Logical Shift Left

DESCRIPTION: Shifts all bits of the operand one place to the left. Bit zero is loaded with a zero. Bit seven is shifted into the carry flag.



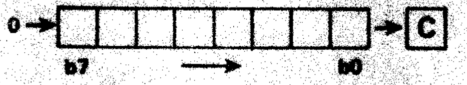
CONDITION CODES:

H: Undefined.
N: Set if bit seven of the result is Set.
Z: Set if all bits of the result are Clear.
V: Set if the carry out of the high order bit is different than the carry into the high order bit.
C: Loaded with bit seven of the original operand.

ADDRESSING MODES: Accumulator, Direct, Indexed, Extended

LSR Logical Shift Right

DESCRIPTION: Performs a logical right shift on the operand. Shifts a zero into the high order bit and bit zero into the carry flag.



CONDITION CODES:

H: Not Affected.
N: Cleared.
Z: Set if all bits of the result are Clear.
V: Not Affected.
C: Loaded with bit zero of the original operand

ADDRESSING MODES: Accumulator, Direct, Indexed, Extended

MARK (ext)	Mark System Stack
DESCRIPTION:	The Mark instruction pushes the specified register list and the user stack pointer onto the system stack, and then loads the user stack pointer from the system stack pointer.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Register List

MUL	Multiply Accumulators
DESCRIPTION:	Multiply the unsigned binary numbers in the A and B accumulators and place the result in the D accumulator.
CONDITION CODES:	H: Not Affected. N: Not Affected. Z: Set if all bits of the result are Clear. V: Not Affected. C: Set if bit seven of the B accumulator is Set.
ADDRESSING MODES:	Inherent

NEG	Negate
DESCRIPTION:	Replaces the operand with its two's complement. Note that 80 (Hex) is replaced by itself and only in this case is overflow set. The value 00 (Hex) is also replaced by itself, and only in this case is carry cleared.
CONDITION CODES:	H: Undefined N: Set if bit seven of result is Set. Z: Set if all bits of result are Clear. V: Set if the original operand was 80 (Hex). C: Cleared if the original operand was 00 (Hex).
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

NOP	No Operation
DESCRIPTION:	This is a single byte instruction that causes the program counter to be incremented. No other registers or memory contents are affected.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Inherent

OR	Inclusive OR
DESCRIPTION:	Performs an eight bit inclusive OR operation between the contents of a register and the memory operand and the result is stored in the register.
CONDITION CODES:	H: Not Affected. N: Set if high order bit of result Set Z: Set if all bits of result are Clear V: Cleared C: Not Affected.
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

ORCC	Inclusive OR into condition code register
DESCRIPTION:	Performs an eight bit inclusive OR operation between the condition code register and the immediate byte and the result is placed in the condition code register. This instruction may be used to set interrupt masks.
CONDITION CODES:	The condition codes are set to the result of the eight bit logical OR of the current condition code bits with he immediate operand. Any condition code bit including the interrupt masks can be set by this operation.
ADDRESSING MODES:	Immediate

PSHS	Push Registers on the System Stack
DESCRIPTION:	Any subset of the MPU registers except the system stack pointer itself are pushed onto the system stack.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Register List

PSHU	Push Registers on the User Stack
DESCRIPTION:	Any subset of the MPU registers except the user stack pointer itself are pushed onto the user stack.
CONDITION CODES:	Not Affected.
ADDRESSIG MODE:	Register List

PULS	Pull Registers from System Stack
DESCRIPTION:	Any subset of the MPU registers except the system stack pointer itself are pulled from the system stack.
CONDITION CODES:	Unaffected unless the condition code register is pulled from the system stack.
ADDRESSING MODES:	Register List

PULU	Pull Registers from the User Stack
DESCRIPTION:	Any subset of the MPU registers except the user stack pointer itself are pulled from the user stack.
CONDITION CODES:	Unaffected unless the condition code register is pulled from the user stack.
ADDRESSING MODES:	Register List

RET (ext) — Return Registers

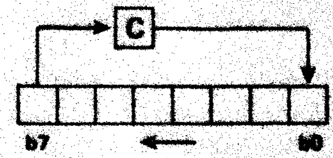
DESCRIPTION: The return instruction pulls the specified register list and the program counter (which effects a return from subroutine) from the system stack.

CONDITION CODES: Not Affected.

ADDRESSING MODES: Register List

ROL — Rotate Left

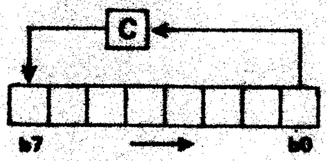
DESCRIPTION: Rotate all bits of the operand one place left through the carry flag. This is a nine-bit shift operation.



CONDITION CODES:
H: Not Affected.
N: Set if the bit seven of the result is Set.
Z: Set if all bits of the result are clear.
V: Set if the bit shifted out of the high order bit is not equal to the bit shifted into the high order bit.
C: Loaded with bit seven of the original operand

ADDRESSING MODES: Accumulator, Direct, Indexed, Extended

ROR	Rotate Right
DESCRIPTION:	Rotates all bits of the operand right one place through the carry flag. This is a nine-bit shift operation.



CONDITION CODES:	H: Not Affected. N: Set if bit seven of result is Set. Z: Set if all bits of result are Clear. V: Not Affected. C: Loaded with bit zero of the original operand.
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

RTI	Return from Interrupt
DESCRIPTION:	The saved machine state is recovered from the system stack and control is returned to the interrupted program.
CONDITION CODES:	Recovered from Stack
ADDRESSING MODES:	Inherent

RTS	Return from Subroutine
DESCRIPTION:	Program control is returned from the subroutine to the calling program. The return address is pulled from the system stack.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Inherent

SBC — Substract with Borrow

DESCRIPTION: Subtracts the contents of memory and the borrow flag from the contents of a register, and places the result in that register.

CONDITION CODES:
H: Undefined.
N: Set if bit seven of the result is Set.
Z: Set if all bits of the result are Clear.
V: Set if the operation causes a two's complement overflow.
C: Set if the operation did not cause a carry from bit seven in the ALU.

ADDRESSING MODES: Immediate, Direct, Indexed, Extended

SCC (ext) — Set Condition Codes

DESCRIPTION: Explicitly sets any subset of the MPU condition flags. This operation is an extended syntax version of ORCC.

CONDITION CODES: All condition flags specified as operands are set. It is not possible to specify the Entire State flag.

ADDRESSING MODES: Condition List

SEX — Sign Extended

DESCRIPTION: This instruction transforms a signed binary eight-bit value in the B accumulator into a signed binary sixteen-bit value in the D accumulator.

CONDITION CODES:
H: Not Affected.
N: Set if the high order bit of the result is Set.
Z: Set if all bits of the result are Clear.
V: Not Affected.
C: Not Affected.

ADDRESSING MODES: Inherent

ST	Store Register Into Memory
DESCRIPTION:	Writes the contents of an MPU register into a memory location.
CONDITION CODES:	H: Not Affected. N: Set if bit seven of stored data was Set. Z: Set if all bits of stored data are Clear. V: Cleared. C: Not Affected.
ADDRESSING MODES:	Direct, Indexed, Extended

SUB	Subtract Memory from Register
DESCRIPTION:	Subtracts the value in memory from the contents of a register.
CONDITION CODES:	H: Undefined. N: Set if the high order bit of the result is Set. Z: Set if all bits of the result are Clear. V: Set if the operation causes a two's complement overflow. C: Set if the operation did not cause a carry from the high order bit in the ALU.
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

SWI	Software Interrupt
DESCRIPTION:	All of the MPU registers are pushed onto the system stack and control is transferred through the SWI vector.
CONDITION CODES:	The IRQ and FIRQ mask bits are set in the condition flag register. All other condition codes are unaffected.
ADDRESSING MODES:	Inherent

SWI2	Software Interrupt 2
DESCRIPTION:	All of the MPU registers are pushed onto the system stack and control is transferred through the SWI2 vector.
CONDITION CODES:	Not Affected.

ADDRESSING MODES:	Inherent
SWI3	Software Interrupt 3
DESCRIPTION:	All of the MPU registers are pushed onto the system stack and control is transferred through the SWI3 vector.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Inherent
SYNC	Synchronize to External Event
DESCRIPTION:	When a SYNC instrucion is executed, the MPU enters a SYNCing state, stops processing instrucitons, and waits on an interrupt. When an interrupt occurs, the SYNCing state is cleared and processing continues. If the interrupt is enabled, the processor will perform the interrupt routine. If the interrupt is masked, the processor simply continues to the next instruction.
CONDITION CODES:	Not Affected.
ADDRESSSING MODE:	Inherent
TFR	Transfer Register to Register
DESCRIPTION:	Transfer a Source register to a destination register. Registers may only by transferred between registers of like size; ie., eight bit to eight bit, and sixteen bit to sixteen bit.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Register

TST	Test the magnitude of an eight bit operand.
DESCRIPTION:	The TST instruction conceptually adds an immediate value of zero to the operand and sets the condition codes accordingly. No data is written to memory or data registers.
CONDITION CODES:	H: Not Affected. N: Set if bit seven of the result is Set. Z: Set if all bits of the result are Clear. V: Cleared. C: Not Affected.
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

6.3.1 - The Star Attribute

The equate directive may be used to define a symbol having the star attribute. This flag is used by the FCB directive to determine whether one or two bytes of data need be generated. The star flag is selected by preceding the equate operand with a crosshatch character, "#". The star flag is also selected if any symbolic reference in the operand expression has its star flag set. An example of the use of the star flag follows:

```
0D0A              CR     EQU   #$0D0A     Set Star Flag
0200                     ORG   $200
0200 4D 53 47     MSG    FCB   "MSG"      Message Text
0203 0D 0A 00            FCB   CR,0       End of Text
```

6.4 - ERR -- Generate an Error

The ERR directive is used to generate an assembler error for documentation purposes. When the ERR directive is encountered, the assembler will generate error number 65, Programmer Signaled Error. This directive may be used to call attention to certain areas of source code, or may be used in conditional assembly in order to detect certain exceptional conditions. For compatability purposes, the mnemonic "FAIL" is also recognised for this operation.

6.5 - ERRIF -- Generate a Conditional Error

The ERRIF directive requires an operand, which it expects to be a truth-valued expression. If the expression value is true, an error message is generated as in the ERR directive, otherwise the ERRIF directive is ignored. Note that the unary truth value operators "?" and "/" may be used to convert an arithmetic expression into a truth value. Some Examples of the ERRIF directive follow:

```
C702                     ERRIF 45<12       False Condition
C702                     ERRIF *=>$C700    Too Much Memory Used
  *** ERROR ***     065 - Programmer Signalled Error
```

6.6 - FAIL -- Generate an Error

The FAIL directive has been provided for compatability with Motorola assemblers and is identical to the ERR directive.

6.7 - FCB -- Form Constant Bytes

The FCB directive is used to define areas of data at assembly time and may have one or more operands, separated by commas. Each operand may be either a character string or an assembler expression. It is important to realize that character strings (such as used in this directive) and character constants are not equivalent. Character constants have a maximum precision of sixteen bits (two characters) while character strings may be of any length.

If an operand begins with one of the assembler quote characters it is considered to be a character string constant. The data generated consists of the ASCII characters enclosed by the quotes. If a quote is to be enclosed within the character string itself, its presence must be indicated by two successive quote characters. For example, the string 'JOHN''S' consists of six characters, and could also be defined by using the alternate syntax "JOHN'S". If character constants are required as part of an expression, they must not be the first term in the expression. Either the unary plus operator can be used to force expression evaluation, or the entire expression can be enclosed in parentheses. An example of this technique is given later.

If the operand does not begin with a quote character, it is considered to be an expression and is evaluated to a sixteen bit word. Null expressions are permitted and have the value zero. The FCB directive normally generates one byte of data for each expression. Double byte significance may be forced by using the significance forcing character, ">" as the first character of the expression and in this case, two bytes of data will be generated. This feature is useful for embedding address words inside of constants.

If an expression operand of the FCB directive has the star attribute, the precision of the expression value is used to determine whether one or two bytes of data will be generated. If the value of the expression is greater than or equal to -128 and less than +128, then only one byte of data is generated. For all other values (including relocatable values) two bytes of data are generated. Note that the star attribute can be overridden through the use of the forcing character "<" to force eight bit significance. Several examples of the FCB directive are presented for clarification:

```
FFFF                     ET  EQU   #-1          Star Flag Set
0D0A                     CR  EQU   #$0D0A       Star Flag Set

0000  00 01 FF               FCB   0,1,-1       Single Bytes
0003  41 42 43 44 45         FCB   "ABCDE"      Character String
0008  41 42 27 43 44         FCB   'AB''CD'     Quote in String
000D  31                     FCB   ('A'-16)     Character Expression
000E  21                     FCB   +"A"-" "     Character Expression
000F  01 F4                  FCB   >500         Forced Significance
0011  F4                     FCB   500          One Byte
0012  41 42 43 0D            FCB   "ABC",13     Two Operands
0016  0F 00 00 0F            FCB   15,,,15      Null Operands
001A  0D 0A FF               FCB   CR,ET        Star Flag Used
001D  0A                     FCB   <CR          Forced Significance
```

If the Motorola compatability option is specified, either on the assembler command line or through the OPT directive, character string constants are supressed. Terms beginning with a quote character are considered to be character constants and are used in expression evaluation. In addition, the significance of each operand is forced to eight bits regardless of the state of the star flag or the forcing characters.

6.8 - FCC -- Form Constant Characters

The FCC directive is used to define character strings in memory. The character string starts with the first non-separator character after the FCC opcode, and terminates with the second occurrance of that character. These delimiters may be any printable ASCII character, and are not considered as part of the character string. Some examples of the FCC directive follow:

```
     000F  41 42 43 44        FCC   /ABCD/    Slash Delimiter
     0013  65 66 67 68        FCC   "efgh"    Quote Delimiter
```

6.9 - FDB -- Form Double Byte

The FDB directive is used to define 16-bit words in memory. It may have one or more operands, separated by commas, and will define one word for each operand expression. This directive is normally used to define addresses.

6.10 - FMB -- Form Multiple Bytes

The FMB directive is used to reserve areas of memory and to initalize them to a single 8-bit value. The first operand of the FMB directive defines the length of the memory area to be defined while the second operand defines the byte of data to be stored in the memory area. The second operand is optional, and if omitted, is assumed to be zero. If the BSZ directive is used, it is processed exactly like the FMB directive and since no second operand is specified, the memory area is properly initialized to zeroes. The first operand of FMB (and of BSZ) must not contain any forward or external references. Some examples of the FMB directive follow:

```
     0019  00 00 00 00 00     FMB   5         Five Bytes
     001E  0A 0A 0A           FMB   3,10      Three Bytes
```

6.11 - LIB -- Library Inclusion

The LIB directive is used to include additional disk files as source language input to the assembler. In effect, the included source file replaces the library directive in the assembly. Only one operand is permitted, and it must have a valid disk file name format. Library inclusion files may be nested, and up to ninety-nine library files may be included in one assembly. If no extension is specified for the library file name, an extension of .TXT is assumed. An example of a library directive used to include a file containing subroutines follows:

```
    6809  17 02 5D    91.         BSR   MOVE      CALL MOVE ROUTINE
                      92.
                      93.         LIB   SUBS      INCLUDE SUBROUTINES
                       1.01  *
                       2.01  .  SUBROUTINES
                       3.01  .
    7069  A6 80        4.01  MOVE LDA   0,X+      GET NEXT BYTE
    706B  A7 A0        5.01       STA   0,Y+      STORE THE BYTE
```

6.12 - NAM -- Provide Module Name

The NAM directive has been provided for source program compatability with Motorola ExBug assemblers. Its use is optional and more than one NAM directive may be used in a program. The operand is a character string up to twelve characters in length and is used as the name of the assembly module. No syntax checking is performed on the operand string. If the print option has been specified, the module name is included on the page headings, otherwise the NAM directive is treated as comments.

6.13 - OPT -- Specify Program Options

The OPT directive is used to change the value of assembler options at assembly time. One or more operands may be specified separated by commas. The assembler recognises both the condensed form of the option name and the complete form, i.e., both "NG" and "NOGEN" are valid option names. The function of each operand is as follows:

G -- The GEN option enables the printing of extra lines of generated code. The default condition of this option is governed by the "G" option specified on the assembler command line.

L -- The LIST option specifies that the assembler generate printed output. This option does not override the "L" option specified on the assembler command line.

M -- The MOTOROLA compatability option supresses non-Motorola extensions to the assembler. Indexed addressing optimization is supressed and branch range checking is selected. Labels are internally truncated to six characters and character string constants in FCB directives are disallowed.

NG --The NOGEN option disables the printing of extra lines of generated code. This option overrides the "G" option specified on the assembler command line.

NL --The NOLIST option disables the generation of printed output. This option is useful for supressing portions of listings.

NP --The NOPAGE option disables the generation of page headings and titles. This option overrides the "P" option specified on the assembler command line. When page headings and titles are disabled, TTL and PAG directives are treated as comments.

P -- The PAGE option enables the generation of page headings and titles, and a new page of output is started. The default condition of this option is governed by the "P" option specified on the assembler command line.

6.14 - ORG -- Set Program Counter Origin

The assembler supports multiple program counter sections, each of which may be absolute or relocatable. The ORG directive is the mechanism used to define a program counter section, and to set its origin. If a label is specified on the ORG statement, this label is used as the name of the program counter section. If no label is specified, the ABSOLUTE program counter is selected. The operand of this directive must not contain forward or external references. Some examples of ORG statements follow:

```
0200                      ORG    $0200       Absolute PC
0400               BUF    ORG    $0400       BUF Program Counter
```

6.15 - PAG -- Start a New Page

The page directive is used to force the assembler to the top of a page of listing. If the page option has not been selected, either by specifying the "P" option on the assembler command line or via the OPT statement, the PAG directive is ignored and appears in the listing. If pagination is in effect, the PAG directive itself disappears from the program listing.

6.16 - PROC -- Begin a Procedure Block

The procedure directive is used to begin a procedure block. If the procedure statement has a label, then that label is used as the name of the procedure dictionary created. If the dictionary has already been defined, then this directive is considered to be a continuation of the previous definiion. A procedure statement with no label creates a dictionary named "*PRnnnn" where "nnnn" is a four-digit number used to make the dictionary name unique. These dictionaries are not printed in the symbol table unless the "U" option has been selected.

Symbols defined in a procedure block are local to that block and cannot be referenced external to the block unless they are explicitly declared as entry definitions by using the colon character ":" as a label terminator. Such symbols appear in the dictionary that contains the procedure and are considered global to the procedure.

In a similar manner, if the procedure block is named, placing a colon on its label will cause the label to be declared as an explicit entry address to the procedure. The value of this label is the value of the currently active program counter. This implies that if multiple program counters are being used with procedure directives, the program counter selection (via USE) should take place prior to procedure definition.

Symbolic references in a procedure are resolved by first searching the current procedure dictionary (called the local dictionary). If the reference has not been resolved, the parent dictionary is searched, and then its parent, and so on, until the global dictionary has been searched. If the reference has still not been resolved, then the public dictionaries are searched.

The ability to declare entry points and to have local labels is of great assistance in writing modular, block structured code. Parameterization can be well defined and controlled by prohibiting access to subroutine temporaries. An example showing the use of procedures and local variables follows:

```
                       1.  *
                       2.  .    COMPARE BYTE FIELD ROUTINE
                       3.  .
                       4.  .    ENTER WITH X => SOURCE FIELD
                       5.  .               Y => TARGET FIELD
                       6.  .               B =  FIELD LENGTH
                       7.  .
                       8.  CMP:  PROC                Declare Entry Point
132F   A6 80           9.        LDA    0,X+         Get Source Byte
1331   A1 A0          10.        CMPA   0,Y+         Compare To Target
1333   26 03          11.        BNE    FAIL         If Not Equal, Exit
1335   5A             12.        DECB                Decrement Count
1336   26 F7          13.        BNE    CMP          Loop Till Done
1338   39             14.  FAIL  RTS                 Exit Routine
                      15.        END
```

Line 8 starts the procedure "CMP" and also causes the symbol "CMP" to be defined as an entry point to the procedure. The procedure is terminated by the end directive on line 15. The label "FAIL" defined on line 14 is a local label and is not defined external to this procedure.

6.17 - PUBLIC -- Begin a Public Dictionary

The PUBLIC directive is used to start a public dictionary. All symbols defined in a public dictionary are available for resolving undefined references. In essence, a public dictionary is considered to be a "parent" to the global dictionary. Symbols defined in a public dictionary are not external references. Procedures or qualified data structures may be defined in a public dictionary, but it is not valid to attempt to define a public dictionary inside of another public dictionary.

If the public statement has a label, that label is used as the name of the public dictionary created. If the dictionary has already been defined, then this directive is considered to be a continuation of the previous definiion. A public statement with no label creates a dictionary named "*PDnnnn" where "nnnn" is a four-digit number used to make the dictionary name unique. These dictionaries are not printed in the symbol table reference unless the "U" option has been selected.

An important consideration in the use of public dictionaries is the order in which symbolic references are resolved. The search for the symbol begins at the current local dictionary and then proceeds through each successive parent dictionary until the global dictionary has been searched. Once the entire path from the local dictionary up through the global dictionary has been searched, then and only then are the public dictionaries searched.

The order in multiple public dictionaries are searched is not defined. More precisely, public dictionaries are not in general searched in the same order in which public directives appear in the input stream. If duplicate labels appear in one or more public dictionaries, which one will be used to finally resolve the reference is unpredictable. An example of the use of a public dictionary is shown:

```
                         28.          PUBLIC               Start Library
CD03                     29.  WARMS   EQU    $CD03         Warm Start Addr
CD24                     30.  PCRLF   EQU    $CD24         Do CR/LF

                         86.          END

0209  BD CD 24           91           JSR    PCRLF         Do Line Feed
020C  7E CD 03           92.          JMP    WARMS         Back to DOS
```

6.18 - QUAL -- Begin a Qualified Data Block

The QUAL directive is used to start an internal qualified data dictionary. The directive must have a label field which is used as the name of the qualified dictionary. Symbols defined inside of a qualified data structure must be referenced by using the name of the symbol qualified by the structure name. An example of a qualified structure and its references is shown:

```
CF69  B6 03 55           52.          LDA    RECORD.SEX    Get Sex Value
CF6C  81 46              53.          CMPA   #"Y"          See if Available
                                 .          .          .

                         82.  RECORD  QUAL                 Start A Structure
0355                     83.  NAME    RMB    10            Name Field
035F                     84.  SEX     RMB    1             Sex Field
                         85.
                         86.          END
```

6.19 - RMB -- Reserve Memory Bytes

The RMB directive is used to reserve a block of memory bytes. No initialization is performed on the reserved memory. The operand of RMB must not contain any forward or external references, and specifies the length of the block of memory to be reserved. Since no object code is generated, RMB directives do not require any space in the object code file. Some examples of RMB follow:

```
0100                SBUF    RMB    256       Small Buffer
0200                LBUF    RMB    10000     Large Buffer
```

6.20 - SETDP -- Set Direct Page Pseudo Register

The SETDP directive is used to inform the assembler of the presumed contents of the direct page register. This value is used by the assembler to decide whether direct or extended absolute addressing should be generated. The most significant eight bits of the expression field is used for addressing calculations. The expression field must have the relocation attribute of absolute. In addition, it must not contain forward or external references.

It is important to realize that the SETDP directive in no way affects the actual contents of the direct page register. It is the programmer's responsibility to insure that the proper values are loaded into this register at execution time. The directive affects only the assemblers addressing mode decision process. An example of the use of the SETDP directive follows:

```
E004           12.  ACIA   EQU   $E004      Interface Address

0204  86 E0    23.         LDA   ACIA>>8    Load Addr MSP
0206  1F 8B    24.         TFR   A,DPR      Load Direct Page

E000           25.         SETDP ACIA       Tell Assembler

0208  96 04    29.  WAIT   LDA   ACIA       Get Status
020A  44       30.         LSRA             Check Receiver
020B  24 FD    31.         BCC   WAIT       Loop if Nothing
020D  96 05    32.         LDA   ACIA+1     Get Data Byte
```

If the Motorola compatability option has been set (either on the command line or via the OPT directive), the SETDP directive uses the least significant eight bits of the value to set the direct page pseudo register. If the most significant eight bits are not zero, a warning is generated.

6.21 - SPC -- Space Listing

The space directive is used to generate blank lines on the assembler listing. If the operand is not specified, one blank line is generated. If the operand is specified, and is positive, it specifies the number of blank lines to be generated. If the operand is negative, it specifies the number of lines that must remain on the page. If less than this number of lines remain, a new page is generated. Some examples of the SPC directive follow:

```
SPC           1 Blank Line
SPC    3      3 Blank Lines
SPC    -10    10 Lines Left
```

6.22 - TTL -- Title

The title directive is used to specify a title for the assembler listing. If the page option has not been selected either by specifying the "P" option on the assembler command line or via the OPT directive, the TTL directive is ignored. When the page option is active, the printing of the actual TTL statement iteself is supressed, a new page of listing is started, and the operand field of the directive is used as the new page heading.

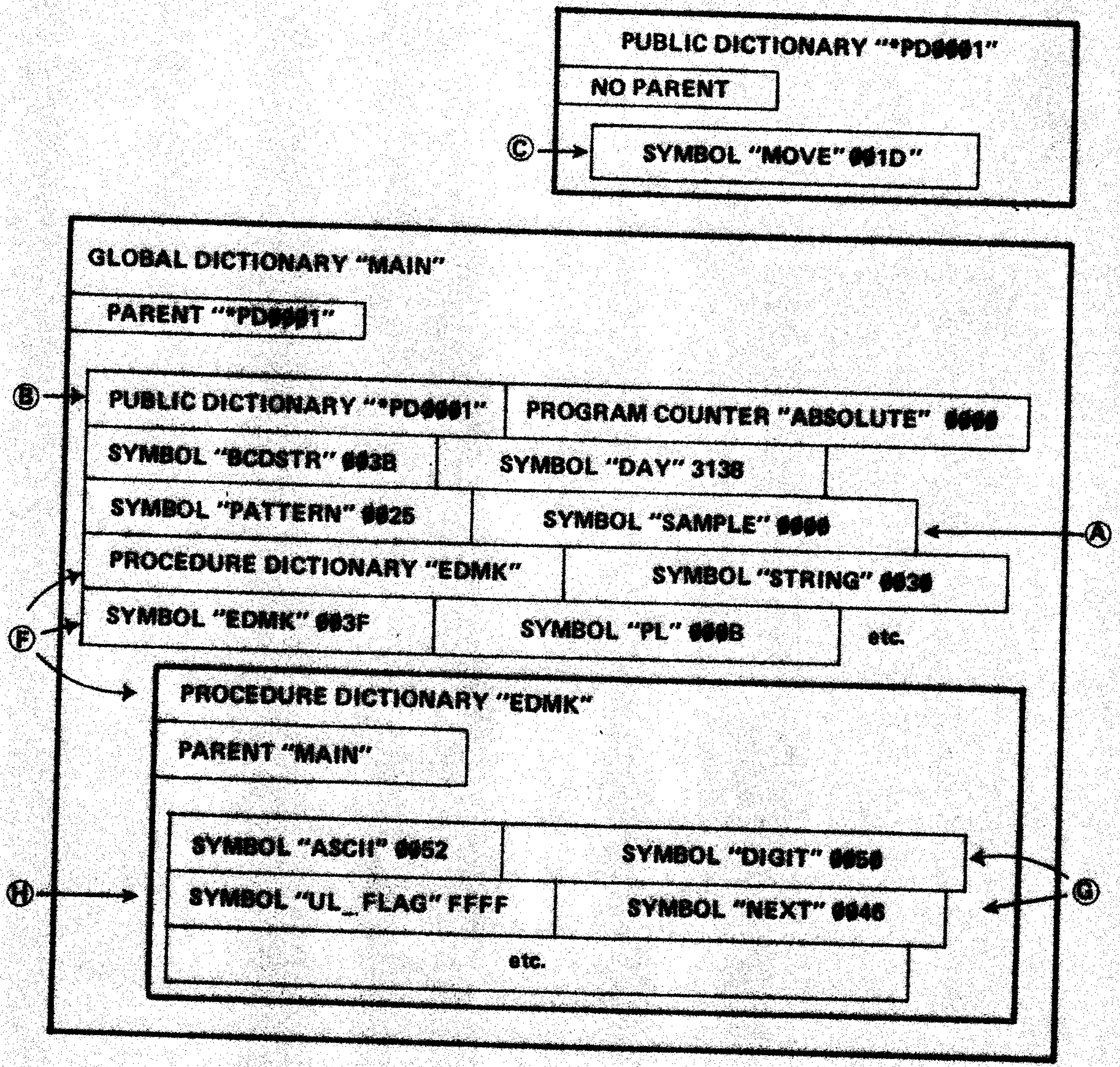
6.23 - USE -- Use Program Counter Section

The USE directive causes the assembler to select a new program counter section. No label may be specified on a use statement, and the operand field must either be a single asterisk, "*", or the name of a program counter section. If a name is specified, the current program counter section is made the previous program counter section, and the named section then becomes the currently active program counter. If an asterisk is specified, the previous program counter section is made the currently active section. If there is no previous section, an error message is produced and the ABSOLUTE program counter is selected.

Multiple program counters can be used to place logically connected pieces of program into different memory locations. In the following example, a pure code procedure references a static data area. The two sections are kept distinct in memory by using multiple program counters. Notice that the references to the data section all utilize direct addressing.

```
                    *
                    .    Pack a Word String
                    .
241A                          USE    CODE
241A  8E 00 2D      PACK      LDX    OORD        Point to Word
241D  1F 12                   TFR    X,Y          Get Two Pointers
241F  D6 2C                   LDB    LEN          Then Get Length
2421  0F 2C                   CLR    LEN          Zero New Length
2423  A6 80         CRAM      LDA    0,X+         Get Word Character
2425  27 04                   BEQ    LOOP         If Null, Bypass
2427  A7 A0                   STA    0,Y+         Store Character
2429  0C 2C                   INC    LEN          Increment New Len
242B  5A            LOOP      DECB                Decrement Count
242C  26 F5                   BNE    CRAM         And Loop
242E  39                      RTS

002C                          USE    DATA         Set Data Section PC
002C                LEN       RMB    1            Word Length
002D                WORD      RMB    32           Buffer for Word
242F                          USE    *            Go Back to Code PC
```



Sample Program
Assembler Symbol Table

The sample program shown includes the subroutine "EDMK" that is used to convert from BCD decimal numbers (packed decimal) to display format, complete with insertion of commas, dollar signs, decimal points, and whatever. This program illustrates several of the capabilities of the SWTPC assembler.

The label SAMPLE at "A" defines an ordinary symbol and appears in the dictionary MAIN along with its value, 0000. A small subroutine MOVE has been included, and placed in the public dictionary via the PUBLIC statement at "B". The label MOVE referenced at "C" defines an ordinary symbol found in the dictionary MAIN.*PD0001. The subsequent end statement at "D" terminates the public dictionary. The subroutine EDMK is included via the LIB statement at "E". The PROC statement at "F" defines a procedure dictionary named MAIN.EDMK and a symbol EDMK in the global dictionary. The symbol definition represents an explicit entry address as denoted by the colon on the label. Labels within the procedure EDMK are local to the procedure and appear only in dictionary MAIN.EDMK. Some examples of local labels are DIGIT and NEXT shown at "G". One additional interesting point is the local variable UL_FLAG that is created on the stack by subroutine EDMK. This variable is referenced via a negative offset from the user stack pointer, as shown at "H".

```
EN                PRINT ALL DATA

ATTERN INTO STRING BUFFER

PATTERN         GET PATTERN                         Ⓐ
STRING          POINT TO OUTPUT STRING
PL              LOAD PATTERN LENGTH
OVE             AND MOVE IN THE PATTERN             Ⓒ

MK ROUTINE TO CONVERT THE STRING

BCDSTR          POINT TO BCD STRING
STRING          AND AT THE OUTPUT STRING
PL              SET THE PATTERN LENGTH
" "             FILL CHARACTER IS A SPACE
DMK             CONVERT THE STRING

F A LEADING DOLLAR SIGN INTO THE STRING

"$"             STUFF IN A DOLLAR SIGN
,-Y             STORE INTO OUTPUT STRING
                *** END OF TEST ***

OVE             INCLUDE MOVE ROUTINE

G

 => SOURCE STRING
 => TARGET FIELD
 =  LENGTH TO MOVE
                                                    Ⓑ
,X+             GET THE SOURCE BYTE
,Y+             STUFF INTO THE TARGET
                DECREMENT THE COUNT
OVE             LOOP UNTIL DONE
                THEN RETURN
                                                    Ⓓ

TERN TO LOOK LIKE  " ddd,ddd.dd"

1E              DIGIT SELECTOR BYTE
1F              SIGNIFICANCE STARTER
 ",DS,DS,DS,",",DS,DS,SS,".",DS,DS


-PATTERN        LENGTH OF PATTERN
L               STRING TO HOLD OUTPUT
00,$23,$03,$12

DMK             INCLUDE SUBROUTINE                  Ⓔ

TO ASCII UNDER CONTROL OF A PATTERN

 =  FILL CHARACTER
 =  PATTERN LENGTH
 => BCD STRING
 => PATTERN STRING

 =  UNCHANGED
 =  ZERO
 => AFTER BCD STRING
 => FIRST SIGNIFICANT DIGIT


. VARIABLES AS OFFSETS INTO THE STACK

                                                    Ⓕ
                FIRST SIG DIGIT POINTER
                BCD STRING POINTER
                REMAINING PATTERN LENGTH
                FILL CHARACTER VALUE
-1              UPPER/LOWER DIGIT FLAG              Ⓗ
```

```
                    *
                    .   CREATE THE PARAMETER LIST ON THE STACK
                    .
003F  34 76 33 E4               MARK  A,B,X,Y        MARK INPUT STACK
0043  5F                        CLRB                 CLEAR SIGNIFICANCE FLAG
0044  6F E2                     CLR   0,-S           CREATE UPPER/LOWER FLAG
                    *
                    .   GET THE NEXT PATTERN CHARACTER AND CONTINUE
                    .
0046  A6 A4         NEXT        LDA   0,Y            GET PATTERN BYTE
0048  81 1F                     CMPA  #$1F           DECIDE WHAT TYPE OF BYTE
004A  22 06                     BHI   ASCII          IF HIGHER, ASCII CHARACTER  Ⓖ
004C  25 02                     BLO   DIGIT          IF LOWER, REGULAR DIGIT
004E  8D 2A                     BSR   SETSIGF        IF EQUAL, SET SIGNIFICANCE
0050  8D 0F         DIGIT       BSR   NEXTBCD        THEN GET THE DIGIT VALUE
                    *
                    .   STORE ASCII CHARACTER OR FILL CHARACTER IN PATTERN
                    .
0052  5D            ASCII       TSTB                 TEST SIGNIFICANCE FLAG
0053  26 02                     BNE   STORE          IF NON-ZERO, STORE DIGIT
0055  A6 C4                     LDA   FILLCHR,U      PICK UP THE FILL CHARACTER
0057  A7 A0         STORE       STA   0,Y+           STORE INTO THE PATTERN
0059  6A 41                     DEC   LENGTH,U       DECREMENT PATTERN LENGTH
005B  26 E9                     BNE   NEXT           IF NON-ZERO, CONTINUE
005D  32 C4 35 F6               EXIT  A,B,X,Y
                    *
                    .   GET NEXT DECIMAL DIGIT FROM BCD STRING
                    .
0061  AE 42         NEXTBCD     LDX   BCD_PTR,U      GET POINTER TO BCD STRING
0063  A6 80                     LDA   0,X+           FETCH TWO BCD DIGITS
0065  63 5F                     COM   UL_FLAG,U      COMPLIMENT HIGH-LOW FLAG
0067  27 06                     BEQ   LOW_DIG        IF ZERO, LOW ORDER DIGIT
0069  44 44 44 44               LSRA;LSRA;LSRA;LSRA  SHIFT RIGHT FOUR BITS
006D  20 04                     BRA   DIGCHEK        AND ENTER CHECK ROUTINE
006F  AF 42         LOW_DIG     STX   BCD_PTR,U      STUFF POINTER BACK
0071  84 0F                     ANDA  #$0F           STRIP OFF HIGH ORDER DIGIT
                    *
                    .   SET SIGNIFICANCE IF NON-ZERO, CONVERT DIGIT TO ASCII
                    .
0073  27 02         DIGCHEK     BEQ   DIGCHAR        IF ZERO DIGIT, BYPASS
0075  8D 03                     BSR   SETSIGF        CHECK DIGIT SIGNIFICANCE
0077  8A 30         DIGCHAR     ORA   #"0"           FORCE AN ASCII DIGIT
0079  39                        RTS
                    *
                    .   SET SIGNIFICANCE AND FIRST SIGNIFICANT DIGIT ADDRESS
                    .
007A  5D            SETSIGF     TSTB                 CHECK SIGNIFICANCE FLAG
007B  26 04                     BNE   SIGEXIT        EXIT IF FLAG ALREADY SET
007D  5C                        INCB                 SET SIGNIFICANCE FLAG
007E  10 AF 44                  STY   FSD_PTR,U      SET FIRST SIGNIFICANT DIGIT
0081  39            SIGEXIT     RTS
                                END

0000                            END   SAMPLE         END OF MAIN PROCEDURE


-- NO ERRORS THIS ASSEMBLY.


-- Procedure:   MAIN

    *PD0001 - PD    0000 ABSOLUTE - PC   003B BCDSTR         3138 DAY - S,P,U
001E DS             003F EDMK                 EDMK - PR      0000 FALSE - P,U
3132 MONTH - S,P,U  0025 PATTERN         000B PL             0000 SAMPLE
001F SS             0030 STRING          0001 TRUE - P,U     3739 YEAR - S,P,U


--     Public:   MAIN.*PD0001

001D MOVE


-- Procedure:   MAIN.EDMK

0052 ASCII          0002 BCD_PTR         0077 DIGCHAR        0073 DIGCHEK
0050 DIGIT          0000 FILLCHR         0004 FSD_PTR        0001 LENGTH
006F LOW_DIG        0046 NEXT            0061 NEXTBCD        007A SETSIGF
0081 SIGEXIT        0057 STORE           FFFF UL_FLAG
```

8.0 - Assembler Error Messages

The assembler performs extensive error checking while processing source input. The philosophy behind the error handler is that the assembler should assist the programmer in staying out of trouble. Of course, no assembler can make a programmer write perfect programs, but it can help by detecting as many faults as possible, and by making unclean coding practices difficult. To this end, the assembler forces certain conventions, such as requiring procedure entry points to be explicitly declared and prohibiting data references into non-structure dictionaries. The error messages produced by the assembler are to a large extent self explanatory, with detailed descriptions of the errors involved documented in this manual.

8.1 - Message Format

Each error message includes a column number, which is the column that the statement scanner was looking at when the error was detected. This column number normally points at the end of the symbol or expression found to be erroneous, however, in certain exceptional cases, the column counter may be several columns different than the item in error.
In the description of the error messages, the term "pointed item" refers to the item preceeding the column counter. Assembler errors have been divided into five classes: Notes, Cautions, Warnings, Errors, and Disasters, in increasing order of serverity.

8.2 - Notes

Notes are produced when the assembler has processed a valid but unlikely operation. For example, it is entirely valid to use a conditional branch instruction with an offset of zero. Since this results in two identical branch paths, the assembler considers the operation unlikely. In this case, the note message "Branch Offset is Zero" is produced.

8.3 - Caution Messages

Cautions are produced when the assembler has detected a condition that may produce unexpected side effects. These messages normally occur for cross assembled 6800 Family instructions. For example, the 6800 instruction CBA (Compare B to A) does not generate a memory reference when run on a 6800 processor while the cross assembled instructions make use of one level of the system stack. The caution "Implicit Use of System Stack" is produced.

8.4 - Warning Messages

Warning messages are produced when a condition has been detected that may produce invalid results. For example, if an instruction causes the generation of an eight bit value and the expression supplied for that value has more than eight bits of significance, it may or may not constitute an error. In this case the warning message "Immediate Value Truncated" would be produced.

8.5 - Error Messages

Error messages are produced when the assembler has detected an invalid but not necessarily fatal condition. For example, if an instruction references an undefined symbol, the code produced for this statement is certainly invalid but other statements are unaffected. In this case the error message "Undefined Symbol Referenced in Expression" would be produced.

8.6 - Disaster Messages

Disaster messages are produced when the assembler has detected a condition which will cause code subsequently produced to be invalid. For example, if the symbol table overflows available memory, all subsequent labels will remain undefined and not be placed in the symbol table. There is little chance of the produced code being anywhere near correct. The disaster message "Insufficient Memory to Define Symbol" is produced.

The following table is a complete list of the error messages produced by the assembler. They have been listed in numerical order of error and are not necessarily grouped by function or cause.

1 -- Error - Undefined Mnemonic Operation Code

The assembler could not find the pointed item in either its mnemonic table or in the macro directory. Five no-op instructions are generated in lieu of the intended code.

2 -- Error - Previously Defined Symbol

The label on the current statement has already been defined in the current dictionary. The previous definition is used and the current definition is suppressed.

3 -- Error - Invalid Register Designator

The assembler expected the pointed item to be a register specification or a register equate value.

4 -- Error - Two Register Specifications Required

Two register specifications are required for transfer and exchange instructions. The assembler encountered a delimiter before the second register specification.

5 -- Error - Invalid Element in Expression

The assembler was parsing an expression when it encountered something that is not an operator, an identifier, a literal, or a valid terminator.

6 -- Error - Undefined Symbol Referenced in Expression

The pointed item is not defined in either the current dictionary or any of its parent dictionaries.

7 -- Error - Unmatched Quotes

The assembler encountered a terminator character before finding the closing quote on a string expression.

8 -- Error - Required Operands Missing

The assembler was processing a mnemonic operation code that requires a memory operand when it encountered a terminator character.

9 -- Error - Immediate Addressing Mode Invalid

Immediate mode addressing was specified on a machine operation that does not support immediate addressing.

10 -- Error - Label Required for This Operation

The current statement contains an assembler directive that requires a label and none has been specified.

11 -- Error - Operand Required for This Operation

The current statement contains an assembler directive that requires an operand and none has been specified.

12 -- Error - Invalid Terminator for Indirection

The assembler was attempting to process an operand specifying indirect addressing when a terminator character was encountered prior to the closing indirection bracket.

13 -- Error - Registers Not Same Size

A transfer or exchange instruction specified two registers that were not both eight bit or both sixteen bit registers.

14 -- Error - Forced Significance Invalid in Immediate Mode

An operand specifying immediate addressing also specified a forced significance. Immediate mode significance is implicit with the instruction being processed.

15 -- Warning - Direct Reference May Be Invalid

An operation using an absolute addressing mode has an operand that forces direct addressing. The assembler has determined that extended addressing is required to reach target address.

16 -- Error - Index Base Register Required

Indexed addressing was specified, but no register designator followed the comma.

17 -- Error - Predec Invalid with PCR Indexing

Indexed addressing was specified using the register predecrement mode, but the register designator specifies the program counter.

18 -- Error - Predec Invalid with Accumulator Offset Indexing

Indexed addressing was specified using the register predecrement mode, but an accumulator offset was specified.

19 -- Error - Predec and Postinc Invalid when Specified Together

Indexed addressing was specified using both the register predecrement mode and the register postincrement mode.

20 -- Error - Postinc Invalid with Register Offset Indexing

Indexed addressing was specified using register postincrement mode, but a register offset was specified.

21 -- Error - Predec or Postinc Require Zero Offset

Indexed addressing was specified using either predecrement or postincrement mode, but the offset value was non-zero.

22 -- Error - Forward Reference Invalid for RMB

The length expression of an RMB directive contained a forward reference. Since this reference may be affected by the outcome of the RMB directive, this constitutes an invalid circular definition.

23 -- Error - Unmatched Quotes in String Constant

The string constant in an FCB directive did not have a closing quote character.

24 -- Error - Expected Operand Not Found

An FDB directive was specified with no operand expressions.

25 -- Error - Unexpected Terminator Encountered

The assembler was processing an FCB directive and encountered a terminator character instead of an expression or string field.

26 -- Caution - Implicit Use of System Stack

A 6800 mnemonic has been encountered and cross-assembled, however, the 6809 code assembled makes implicit use of the system stack, which the 6800 code does not.

27 -- Error - Label Field Invalid for This Operation

The current statement contains an assembler directive that cannot process a label and one has been specified.

28 -- Error - End of Line Before Terminating Character

The assembler was processing an FCC directive and encountered the end of the current input line before locating the matching terminating character.

29 -- Error - Indexing Specified with Immediate Addressing

Immediate addressing mode was specified, but an index register designator was located or implied by the expression.

30 -- Warning - Maximum Negative Number Negated to Zero

While the assembler was evaluating an expression, a maximum negative number was negated. The resuling two's compliment overflow forced a zero result.

31 -- Error - Operator Stack Overflow

The current expression contains operators nested too deep for the assembler evaluator to parse.

32 -- Error - Value Stack Overflow

The current expression has too many terms for the assembler evaluator to parse.

33 -- Error - Operator Encountered Out of Context

The assembler was parsing an expression and expected a value token when an operator or terminator was encountered.

34 -- Error - Missing Right Parentheses in Expression

The pointed expression has more left parentheses than right parentheses.

35 -- Error - Too Many Right Parenthesis in Expression

The pointed expression has more right parentheses than left parentheses.

36 -- Error - Invalid Binary Operator in Expression

The pointed character string was encountered in the context of a binary operator and is not a valid binary operator.

37 -- Error - Invalid Unary Operator in Expression

The pointed character string was encountered in the context of a unary operator and is not a valid unary operator.

38 -- Warning - Use of WAI is not Equivalent to CWAI

A 6800 WAI Mnemonic has been encountered and cross assembled into a conditioned wait instruction. Note that the 6800 instruction sequence NOP; CLI; WAI; should be replaced with the 6809 instruction CWAI $EF. The cross-assembled sequence of instructions can result in an interrupt occurring after the execution of the CLI instruction but before the WAI.

39 -- Disaster - Insufficient Memory to Define Symbol

The assembler found that there was insufficient memory to insert the label of the current statement into the symbol table.

40 -- Disaster - Insufficient Memory for Library Inclusion

The assembler found that there was insufficient memory to open the library inclusion file.

41 -- Error - Library File Could Not Be Opened

The specified library file did not exist on disk, or there was a directory error resulting in failure of the open on the library file.

42 -- Error - Library File Specification Invalid

The file specification on the library statement was invalid.

43 -- Error - Library File Specification Required

The library statement requires a file specification.

44 -- Warning - Library Inclusion Numbers May Be Invalid

The assembler will allow any number of inclusion files, however, the inclusion count is two decimal digits long. If more than 99 inclusions are used, the inclusion number will no longer be valid.

45 -- Warning - Multiply Caused Two's Complement Overflow

The product of two sixteen bit numbers could not be contained in the sixteen bit result. The least significant sixteen bits of the product was used as the result.

46 -- Warning - Divide by Zero

The divisor value was found to be zero and the assembler has substituted the maximum positive number as the result.

47 -- Error - Absolute Value Required by ORG

The ORG directive forces the program counter to become absolute hence the value of the expression on the ORG statement must have a relocatability attribute of absolute.

48 -- Error - Absolute Value Required by RMB

The RMB directive requires a non-relocatable length for its operand.

49 -- Caution - E Condition Flag Undefined Except on Stack

The ANDCC or ORCC operation referenced the "E" condition flag which is not meaningful except on the stack.

50 -- Error - Attempt to Redefine a Protected Symbol

The current statement attempts to redefine the value of a protected symbol.

51 -- Warning - Label Subsequently Redefined

This is the first occurance of a label that is subsequently redefined.

52 -- Disaster - Code Generation Pass Phasing Error Detected

This error indicates an internal malfunction in the assembler and should be reported at once to Southwest Technical. Copies of the program generating this error along with all relevent data should be included with the report.

53 -- Error - Undefined Node Referenced in Structure

The assembler was processing a qualified data name when it encountered a node name that was not found in the structure's dictionary.

54 -- Error - Null Node Name Invalid in Structure

The assembler was processing a qualified data name when it encountered a null symbol or terminator character.

55 -- Disaster - Insufficient Memory to Define Procedure

Insufficient memory remained in the assembler dictionary space to allocate space for a new procedure or data dictionary and the new dictionary has not been defined.

56 -- Disaster - Insufficient Memory to Define Program Counter

Insufficient memory remained in the assembler dictionary space to allocate a new program counter.

57 -- Error - Insufficient Memory for Data Dictionary

Insufficient memory remained in the assembler dictionary space to allocate space for a new qualified data dictionary and the new qualified name has not been defined.

58 -- Error - Eight Bit Index Offset will be Insufficient

Constant offset indexed addressing was specified with the offset forced to eight bits. The assembler has determined that this offset will be insufficient to allow the instruction to reach its target.

59 -- Error - Forced Short Branch cannot Reach Target

The expression field of a branch instruction has forced the offset to eight bits. The assembler has determined that this offset is insufficient to allow the instruction to reach its target.

60 -- Error - Branch Out of Range

The assembler range-check option was specified and the current short branch cannot reach its target.

61 -- Error - Absolute Value Required for FMB

The FMB directive requires a non-relocatable length for its first operand.

62 -- Error - Forward Reference Invalid for FMB

The length expression of an FMB directive contained a forward reference. Since the reference may be affected by the outcome of the FMB directive, this constitutes an invalid circular reference.

63 -- Error - Sixteen Bit Precision Cannot be Forced for FMB

The value expression of an FMB directive has a forced sixteen bit precision. The assembler has used the low-order eight bits of the expression as the fill bytes.

64 -- Error - Invalid Option Specified

The specified option is invalid for the OPT directive and has been ignored.

65 -- Error - Programmer-Signalled Error

An error message has been generated by an ERR or ERRIF conditional assembly statement.

66 -- Error - Specified Register Invalid as Index Base

Indexed addressing mode was specified but the register specified as the index address base cannot be used for indexed addressing.

67 -- Error - Operand Required for Indirection

The assembler was processing an operand specifying indirect addressing when it encountered and end of statement operator. Null operands are not valid when using indirect addressing.

68 -- Warning - Immediate Value Truncated

The assembler has truncated significant bits from a sixteen bit expression to obtain the required eight bit immediate value.

69 -- Error - Leading Bracket Required for Indirection

The pointed expression has terminated with a closing right bracket indicating indirect addressing mode, but no leading left bracket preceeded the expression.

70 -- Warning - No Registers Specified in List

A push or pull operation was specified with a null register list.

71 -- Error - Parent Reference Invalid in Global Dictionary

An explicit parental reference was encountered while processing code in the global dictionary where only local or global references are valid.

72 -- Error - Directive Requires Primary Statement

An assembler directive was encountered in the context of a secondary statement. Directives must always be specified as primary statements.

73 -- Warning - Absolute Program Counter Selected

Too many levels of USE Previous were specified. The default absolute program counter has been selected.

74 -- Error - Invalid Program Counter Specified

A USE statement specifies an identifier that has not been previously defined as a program counter name.

75 -- Note - Branch Offset is Zero

A branch instruction was encountered with a computed offset value of zero. If full optimization was selected, the entire branch instruction is supressed.

76 -- Warning - Missing End Statement

The assembler has detected an end of file condition prior to processing the end statement for the MAIN procedure. No transfer address has been assigned to the object code module.

999 -- Disaster - Invalid Error Address

This error indicates an internal malfunction in the assembler and should be reported at once to Southwest Technical. Copies of the program generating this error along with all relevent data should be included with the report.