# SECTION 5
# THE AIM 65 ASSEMBLER

The process of translating individual computer program instructions written in mnemonic or symbolic form (the source program) into actual processor instructions in machine code form (the object program) is called an assembly. The computer program that performs this translation is an Assembler. The mnemonics and symbols used in writing the source program, commonly called the source code, is called the assembly language. One assembly language instruction translates into one processor instruction. The object program is commonly referred to as the object code.

The optional AIM 65 Assembler is a ROM resident, two-pass assembler. It is supplied as one 4K R2332 ROM that plugs into socket Z24 on the AIM 65 Master Module. The assembler allows both instruction and data addresses to be specified symbolically rather than requiring absolute addresses. During Pass 1 the Assembler determines the values of the symbols. The symbols and their corresponding values are placed in a symbol table for use during Pass 2. The assembler generates the actual object code during Pass 2, using the symbol values from the symbol table to generate addresses and displacements and to create data values. Extensive error checking is performed during Pass 2 to determine if the instructions have been correctly coded. Any detected errors are displayed/printed. The assembly listing is also generated during Pass 2.

To use the Assembler, source code is first prepared using the Editor then stored on audio cassette tape, or passed directly from the Text Buffer in RAM to the assembler. If a teletype is used, the source code may be stored on punched paper tape and read back as the input to the Assembler.

Operation of the Assembler is completely automatic within each pass once you specify such information as where the input source code is coming, where the output object code is to be directed, if a full assembly listing or just an errors-only listing is to be generated.

Assembler directives included in the source code provide additional and overriding instructions for listing generation. When audio cassette tape is used as input, a file linkage capability allows multiple files to be used.

## 5.1 ASSEMBLER ROM INSTALLATION

Before removing the ROM from its shipping package, be sure to observe the handling precautions in Section 1.4. Turn off power to the AIM 65. Remove the ROM from the shipping package. Inspect the pins to be sure they are straight and free from any shipping or protective foam material. Insert the ROM into socket Z24, being careful to observe the proper device orientation. (See Figure 1-1). Support the Master Module from underneath the socket after the pins are aligned and started into the socket. Press firmly and evenly on the ROM until it is inserted completely into the socket.



## 5.2 THE SYMBOL TABLE

Working space in RAM must be allocated for symbol table storage during Pass 1. Eight bytes of RAM must be allocated for each uniquely defined symbol in the source code; six bytes for the symbol itself and two bytes for the symbol value. Since each symbol requires eight bytes of memory, the end of the symbol table will be a multiple of eight bytes from the starting address. If the allocated symbol table area is not large enough, the assembler will terminate Pass 1 after displaying the error message:

SYM TBL OVERFLOW

The starting and ending address limits of the symbol table are entered during assembly Pass 1.

The actual starting address will correspond to the entered starting address limit. The actual ending address will be lower than the ending address limit.

The starting and upper limit addresses of the symbol table and the number of symbols in the table can be determined by examining these memory locations:

| ADDRESS | PARAMETER | EXAMPLE |
|---|---|---|
| $003A | Symbol Table Starting Address Low | $00 |
| $003B | Symbol Table Starting Address High | $03 |
| $003E | Symbol Table Upper Limit Address Low | $FF |
| $003F | Symbol Table Upper Limit Address High | $03 |
| $000B | Number of Symbols (HEX) High | $00 |
| $000C | Number of Symbols (HEX) Low | $12 |

The address of the last symbol can be computed by multiplying the number of symbols in the symbol table by eight and adding the result to the starting address.

For example,

$0300 + ($0012 X 8) = $0300 + $0090 = $0390
(Address of the last symbol)

NOTE

If the source code, object code, and symbol table are all to reside in RAM during the assembly, take care to prevent overwriting the source code with either the symbol table or the object code, or the symbol table with the object code. Extreme care must be taken to avoid overwriting the source code; it will have to be read into the Text Buffer again. It is good practice to periodically save the source code on permanent storage media (e.g., audio cassette) to prevent inadvertent loss due to overwriting, editor initialization, or AIM 65 power loss.

## 5.3 ASSEMBLY CONSIDERATIONS

### 5.3.1 Memory to Memory Assembly

The actual object code addresses can be examined by printing the assembly listing during Pass 2 without directing the object code to memory. The object code can be directed



either to audio cassette tape or to the dummy device (i.e., no output). If the output is directed to audio cassette, it can be then loaded into memory using the Monitor L command without regard to the previously used symbol table locations. If the object code addresses conflict with the source code, the source code should be saved on audio cassette tape before loading the object code.

If no object code output was generated, and examination of the object code addresses on the assembly listing shows no source code or symbol table address conflicts, Pass 2 can be re-run to direct the object code to memory and not to generate an assembly listing (since one was generated during the first Pass 2).

### 5.3.2 Tape To Tape Assembly

A program with many symbols may require a major portion or all of RAM for the symbol table. In this case, the source code should be saved on audio cassette before the assembly. Pass 1 and Pass 2 should both be run with the input from audio cassette. The output object code should be directed also to a different audio cassette (see Section 9 for detailed audio cassette operation). The size of the program is now limited only by the RAM memory available to handle the number of symbols in the source code.

### 5.3.3 User Program Constant Storage

The Assembler uses Page 0 locations 0004 through 00DE and Page 1 locations 0170 through 0183. For this reason, you should not assemble any instructions or constants into these locations when assembling to memory (OBJ? N). User program variables can be assigned to these locations, however, and instructions/constants can be loaded in these locations, after the assembly is complete.

## 5.4 USING THE ASSEMBLER

Use the AIM 65 Assembler as follows:

1. To enter the assembler, type N after display of the Monitor prompt. AIM 65 will respond with:

```
<N>
ASSEMBLER
FROM=Λ
```

2. Enter the symbol table starting address in hexadecimal. Terminate the address by typing RETURN. Typing RETURN without entering a value will cause the starting address to default to its previously entered value. The newly entered or default value will be displayed. For example, if 0300 is entered, AIM 65 will respond with:

```
FROM=0300  TO=Λ
```

CAUTION

Since all of Page 0 is used for assembler variables and Page 1 is reserved for user and AIM 65 Monitor stack usage and for AIM 65 variables, the symbol table starting address must be equal to or greater than 0200.



3. Enter the symbol table ending address in hexadecimal. Terminate the address entry with RETURN. Typing RETURN without entering a value will cause the ending address to default to its previously entered value. The newly entered or default value will be displayed. For example, if 0400 is entered, AIM 65 will respond with:

```
FROM=0300  TO=0400.

IN=Λ
```

4. Enter the code of the input device that contains the source code. Valid options are:

```
M = Text Buffer in memory (RAM)
T = audio cassette tape
L = TTY punched paper tape
U = user defined peripheral
```

CAUTION

If Pass 1 is to be performed from memory (IN = M), be sure that the symbol table addresses do not conflict with the addresses of the source code in the Text Buffer. Part or all of the source code will be overwritten with the symbol table in this case.

Note that the source code will be displayed as it is being read from the input device.

A. If M is entered, AIM 65 will display M. Go to Step 5.

B. If T is entered, AIM 65 will ask for the file name:

IN=T F=^

NOTES

1. In order to use an audio cassette for input, the audio cassette recorder must have a remote control capability, with connections as described in Section 9.1. The Assembler processes source code by operating on a block of 80 bytes at a time. During this time, the cassette recorder is halted using the recorder remote control input after a block of source code is read. The recorder will stop with the cassette tape positioned between blocks of input data. When the assembler has processed the block of 80 bytes, the recorder will be restarted to read another block of data.

2. The source code must have been recorded with the GAP variable in $A408 equal to $80, or larger.

Enter the file name under which the source code was stored. If the file name is less than five digits, end the input with a



NOTES (Cont.)

RETURN or SPACE. AIM 65 will then ask for the audio cassette recorder number. For example:

IN=T F=SRCE1 T=^

Enter the the audio cassette recorder number (1 or 2) from which the source code is to be loaded. End the input with a RETURN or a SPACE. If 1 was entered, AIM 65 will respond with:

IN=T F=SRCE1 T=1

AIM 65 will search for the specified file name. Upon locating a readable tape file, the file name on tape will be compared to the entered file name. If the file names are not identical, AIM 65 will display the search message and block count of the recorded file as the file passes. If file name PROG1 is read, AIM 65 will respond with:

SRCH F=PROG1 BLK=XX Where XX=the block count.

When the entered file name is located on the tape, The Assembler will continue to Step 5.

C. If L is entered, reading of the source code on punched paper tape from the TTY should be initiated as described in Section 9.2.7.

D. If U is entered, Pass 1 will be initiated using the user defined input as described in Section 7.

5. AIM 65 will ask if the total assembly list or just an errors only list is to be displayed/ printed:

```
LIST?^
```

If only errors are to be listed, type N. An errors-only listing will be generated during Pass 2.

If the full assembly listing is to be produced, type Y. The complete assembly listing includes the total source program, reformatted for proper output spacing, the address with each label, the generated object code and any detected errors.

6. AIM 65 will ask where the full assembly or the errors-only listing is to be directed.

```
LIST-OUT=^
```

Type one of the following valid options:

```
RETURN or SPACE  =  Display/Printer
P = Printer
T = Audio Cassette (AIM 65 Source Code Format)
U = User defined
L = TTY (See Section 9.2)
```



7. AIM 65 will then ask where the object code is to be directed:

```
OBJ?^
```

If the object code is to go directly into memory, type N. Go to Step 6.

CAUTION

If the output is to go into memory, be sure the object code addresses do not conflict with source code in the Text Buffer if the input is from memory or with the symbol table addresses.

If the object code is to be directed to an output device and not to memory, type Y. AIM 65 will respond by asking for the output device code.

```
OBJ-OUT=^
```

8. Type one of the following valid option codes:

```
RETURN or SPACE  =  Display/Printer
P = Printer only
T = Audio Cassette (AIM 65 Format)
L = TTY (See Section 9.2)
X = Dummy Device (no output)
U = User Defined
```

NOTE

The selected OBJ-OUT= option must not conflict with the previously selected LIST-OUT option or else both listing and object code output will be directed to the same output device in an intermixed manner.

9. AIM 65 will initiate Pass 1 and display:

PASS 1

During Pass 1, the Assembler creates the symbol table. If the allocated symbol table area is too small to store all the symbols, AIM 65 will display SYM TBL OVRFLOW and return to the Assembler entry point.

10. If Pass 1 is completed successfully, AIM 65 will automatically initiate Pass 2 if the input (IN=) is from memory (M) or user-defined (U). If the input is from audio tape (T) or punched tape (L), the Assembler will halt and display

PASS 2

Rewind the tape and type SPACE to start Pass 2.

11. Pass 2 will be performed. The selected errors-only or full assembly listing will be directed to the LIST-OUT device. The assembled object code wlll be directed to the OBJ?/OBJ-LIST device. Upon completion of Pass 2, control will be returned to the Monitor.



Any error detected during Pass will be identified by a number corresponding to the error code (see Table 5-1) in this format:

**ERROR XX where XX = 01 to 21.

At the completion of the assembly listing the number of detected errors will be reported.

ERRORS= XXXX where XXXX = the error count.

NOTE

Any .OPT LIS, NOL, ERR or NOE directives in the user program will override the user response to the LIST? prompt.

## 5.5 ASSEMBLER EXPRESSIONS

Assembler expressions are very useful tools to facilitate programming and to generate both readable and easily changeable code.

There are two components of assembler expressions: elements and operators.

### 5.5.1 Elements

Elements may be classified into three distinct types: constants, symbols, and the location counter.

Table 5-1. Assembler Error Messages

| | |
|---|---|
| | SYM TBL OVERFLOW<br>During Pass 1, more unique symbols were detected than allowed in the symbol table. The symbol table length allocation can be enlarged by changing either the symbol table starting and/or ending address by re-entering Pass 1. |
| 01 | ** UNDEFINED SYMBOL<br>The assembler has found a symbol in an operand expression which is nowhere defined (as a label or as the destination field of an equate directive) in the source code. This error will also occur if a reserved name (A, X, Y, S, or P) is referenced as a symbol in an expression. |
| 02 | ** LABEL PREVIOUSLY DEFINED<br>The first field on the line, interpreted as a symbol, has been found already defined with a value in the symbol table. A forward reference to a page zero defined symbol has caused a misalignment in address values from Pass 1 to Pass 2. |
| 03 | ** ILLEGAL OR MISSING OPCODE<br>The assembler has found a line containing a label, followed by an expression, which it tried to interpret as an instruction. |
| 04 | ** ADDRESS NOT VALID<br>An address referenced in an instruction or in one of the assembler directives (.BYTE, .WORD, or .DBYTE) is invalid. |



Table 5-1. Assembler Error Messages (Cont.)

| | |
|---|---|
| 05 | ** ACCUMULATOR MODE NOT ALLOWED<br>Following a legal instruction mnemonic and one or more spaces, is the letter A followed by one or more spaces (denoting the accumulator addressing mode). The assembler tried to use the accumulator as the operand. However, the instruction in the statement is one which does not allow reference to the accumulator. |
| 06 | ** FORWARD REFERENCE TO PAGE ZERO<br>An operand expression containing a forward reference has been found. |
| 07 | ** RAN OFF END OF LINE<br>This error message occurs when the assembler is looking for a needed field and runs off the end of the line image before the field is found. |
| 08 | ** LABEL DOESN'T BEGIN WITH ALPHABETIC CHARACTER<br>The first non-blank field, being neither a comment nor a valid instruction, is assumed to be a label. However, the first character of the field begins with a numeric character (0-9), violating the rules of symbol construction. |
| 09 | ** LABEL GREATER THAN SIX CHARACTERS<br>The first field on the line is a string containing more than six characters. Lacking a semicolon prefix, denoting a comment, it is assumed to be a symbol whose length limit has been exceeded. |

Table 5-1. Assembler Error Messages (Cont.)

| | |
|---|---|
| 10 | ** LABEL OR OPCODE CONTAINS NON-ALPHANUMERIC<br>The label or opcode field on a line (illegally) contains a character which is not alphanumeric. |
| 11 | ** FORWARD REFERENCE IN EQUATE OR ORG<br>The expression on the right side of an equal sign contains a symbol that has not been previously defined. |
| 12 | ** INVALID INDEX - MUST BE X OR Y<br>A legal operand expression follows an opcode. Following this expression, is a comma (denoting indexed addressing) and an invalid string where either X or Y was expected. This error will be given whether an indexed addressing mode is legal for the corresponding instruction mnemonic or not. |
| 13 | ** INVALID EXPRESSION<br>While evaluating an expression, the assembler found a character that it could not interpret. |
| 14 | ** UNDEFINED ASSEMBLER DIRECTIVE<br>If a period is the first character in a non-blank field, the assembler interprets the following three characters as an assembler directive. Either an invalid directive has been found or the first three characters of one of the options in the .OPT directive are uninterpretable. |
| 15 | ** INVALID PAGE ZERO COMMAND<br>An invalid page zero indexed operand has been detected, for example STA #20,X. An invalid non-page zero indexed operand, for example STA #20, or an invalid page zero operand without indexing will result in error 18. |



Table 5-1. Assembler Error Messages (Cont.)

| | |
|---|---|
| 17 | ** RELATIVE BRANCH OUT OF RANGE<br>A branch instruction can branch only 127 bytes forward or 128 bytes backward. This error message indicates an out-of-range branch. |
| 18 | ** ILLEGAL OPERAND TYPE FOR THIS INSTRUCTION<br>After finding an instruction mnemonic that does not allow implied addressing, the assembler passes to the operand field and determines what type of operand it is (indexed, absolute, etc.). This error message is printed if the type of operand found is invalid for the instruction. |
| 19 | ** OUT OF BOUNDS ON INDIRECT ADDRESSING<br>An indirect address is recognized as such by the parentheses that surround it in the operand field of an instruction mnemonic. Since indirect addressing requires two bytes of page zero memory, the address referencing this area must be less than or equal to 254. |
| 20 | ** A, X, Y, S, AND P ARE RESERVED LABELS<br>One of the five reserved names (A, X, S, X, and P) has been used as a symbol. |
| 21 | ** PROGRAM COUNTER NEGATIVE - RESET TO 0<br>An attempt to reference a negative memory location will cause this error message, and the Program Counter to be reset to 0. |

### 5.5.2 Constants

Numeric constants may be written in several bases. The base is specified by the type of prefix character preceding the digits, as defined in the following table.

| PREFIX CHARACTER | BASE |
|---|---|
| (none) | 10 (Decimal) |
| $ | 16 (Hexadecimal) |
| @ | 8 (Octal) |
| % | 2 (Binary) |

Some examples are:

```
==0000
        *=$200
==0200
10      BYT $10,10,@
10,%10
0A
08
02
AD10F7 LDA $F710
A51D   LDA 29
A57E   LDA @176
A56D   LDA %01101101
```

#### ASCII LITERAL CONSTANTS

ASCII literal constants, enclosed in quotes, are used to insert the ASCII representation of character strings into memory.



For example:

```
25      .BYT 'M','I''
M','''
49274D
==0211
27
A950   LDA #'P
A927   LDA #'''
A935   LDA #'5
```

Note that two quotes are needed to represent a quote in memory. Thus, in the last field of the .BYT directive, the first represents a single quote, and the last closes off the string.

### 5.5.3 Symbols

Symbols are names used to represent numerical values. They may be from one to six alphanumeric characters long, and the first character must be alphabetic. The 56 valid opcodes (listed in Table 5-2) and the reserved symbols A, X, Y, S, and P have special meaning to the Assembler, and may not be used as symbols.

For example:

```
==0218 VARBLE
       =$2C
==0218 DATA1
AB     .BYT $AB,VARB
LE
2C
==021A LAB199
AD1802 LDA DATA1
A52C   LDA VARBLE
```

### 5.5.4 Location Counter

The location counter, referenced by the character "*", is a sequential counter used by the Assembler to keep track of its current position in memory. It may be freely used in expressions within a program.

For example:

```
021F    .DBY *
A02102 LDA *
```

### 5.5.6 Operators

Two arithmetric operators are allowed in the R6500 assembly language:

| OPERATOR | OPERATION |
|---|---|
| + | Addition |
| - | Subtraction |

Evaluation of expressions proceeds strictly from left to right, with no parenthetical grouping allowed; all operators have equal precedence.

In addition, there are two special operators:

| CHARACTER | OPERATION |
|---|---|
| > | High-Byte Selection |
| < | Low-Byte Selection |



Operators < and > truncate a two-byte value to its high or low byte, respectively.

For example:

```
==0224 HIGH
AB      .BYT >$ABCD,<
*,5+<HIGH-%10
25
27
A541    LDA <*+2>$1AC2
A51A    LDA %101+7+$7
+07
A502    LDA >HIGH-$40
+<65
```

Expressions which evaluate to negative values are illegal. The twos complement representation of a negative number must be expressed as an unsigned (preferably hexadecimal) constant (e.g., write "-1" as "$FF").

Note especially that expressions are evaluated at assembly time, not at execution time.

## 5.6 ASSEMBLER SOURCE STATEMENTS

Assembler source statements are comprised of up to four fields:

```
[label]   [opcode [operand]]   [;comment]
```

Brackets surrounding a field indicate that it is optional. Thus, although none of the fields is mandatory, an opcode field must precede an operand field. Input to the Assembler is free form; any field may start in any column.

In particular, note that due to the reserved opcodes, the user is able to precede labels with spaces. If no

label is present, an opcode may be placed in the first column.

Fields in a statement need only be separated by a single space. If the fields are separated in this manner, the Assembler will columnize the fields and produce a readable listing. The user's program may then be stored on audio cassette in a highly compressed form.

Also note that the comment field should be preceded by a semicolon. If the semicolon is omitted, the comment field will not be placed in its proper column in the listing.

### 5.6.1 Labels

A label is a one- to six-character string of alphanumeric characters. It must begin with an alphabetic character, and must appear as the first field on a line, although it may begin in any column. Using a label is a way to assign the current value of the location counter to the symbol before the rest of the line is processed by the Assembler. Labels are used with instructions as branch targets and with memory data cells for reference in operands.

A line containing only a label is valid, so several labels may be assigned to the same memory location by putting each on a separate line:

```
==0120 SAME1
==0120 SAME2
==0120 SAME3
AD2002 LDA SAME1
```



### 5.6.2 Opcodes and Operands

Two distinct classes of assembler instructions are available to the programmer: machine instructions and assembler directives.

### 5.6.3 Machine Instructions

The 56 valid machine instruction mnemonics (Table 5-2) represent the operations implemented on the R6500 family of microprocessors. When assembled, each mnemonic generates one byte of machine code, the actual bit pattern depending upon both the operation specified in the opcode field and the addressing mode determined from the operand field. The operand field may generate one or two bytes of address.

Table 5-2. R6500 Microprocessor Instruction Set--Alphabetic Sequence

| | | | |
|---|---|---|---|
| | ADC | Add Memory to Accumulator with Carry | |
| | AND | AND Memory with Accumulator | |
| | ASL | Shift Left One Bit (Memory or Accumulator) | |
| | BCC | Branch on Carry Clear | |
| | BCS | Branch on Carry Set | |
| | BEQ | Branch on Result Zero | |
| | BIT | Test Bits in Memory with Accumulator | |
| | BMI | Branch on Result Minus | |
| | BNE | Branch on Result Not Zero | |
| | BPL | Branch on Result Plus | |
| * | BRK | Force Break | |
| | BVC | Branch on Overflow Clear | |
| | BVS | Branch on Overflow Set | |
| * | CLC | Clear Carry Flag | |
| * | CLD | Clear Decimal Mode | |
| * | CLI | Clear Interrupt Disable Bit | |
| * | CLV | Clear Overflow Flag | |
| | CMP | Compare Memory and Accumulator | |
| | CPX | Compare Memory and Index X | |
| | CPY | Compare Memory and Index Y | |
| | DEC | Decrement Memory by One | |
| * | DEX | Decrement Index X by One | |
| * | DEY | Decrement Index Y by One | |
| | EOR | Exclusive-OR Memory with Accumulator | |
| | INC | Increment Memory by One | |
| * | INX | Increment Index X by One | |
| * | INY | Increment Index Y by One | |
| | JMP | Jump to New Location | |
| | JSR | Jump to New Location Saving Return Address | |
| | LDA | Load Accumulator with Memory | |
| | LDX | Load Index X with Memory | |
| | LDY | Load Index Y with Memory | |
| | LSR | Shift Right One Bit (Memory or Accumulator) | |
| * | NOP | No Operation | |
| | ORA | OR Memory with Accumulator | |
| * | PHA | Push Accumulator on Stack | |
| * | PHP | Push Processor Status on Stack | |
| * | PLA | Pull Accumulator from Stack | |
| * | PLP | Pull Processor Status from Stack | |
| | ROL | Rotate One Bit Left (Memory or Accumulator | |
| | ROR | Rotate One Bit Right (Memory or Accumulator) | |
| * | RTI | Return from Interrupt | |
| * | RTS | Return from Subroutine | |
| | SBC | Subtract Memory from Accumulator with Borrow | |
| * | SEC | Set Carry Flag | |
| * | SED | Set Decimal Mode | |
| * | SEI | Set Interrupt Disable Status | |
| | STA | Store Accumulator in Memory | |
| | STX | Store Index X in Memory | |
| | STY | Store Index Y in Memory | |
| * | TAX | Transfer Accumulator to Index X | |
| * | TAY | Transfer Accumulator to Index Y | |
| * | TSX | Transfer Stack Pointer to Index X | |
| * | TXA | Transfer Index X to Accumulator | |
| * | TXS | Transfer Index X to Stack Pointer | |
| * | TYA | Transfer Index Y to Accumulator | |

*Instructions legal only in the implied addressing mode.



## 5.7 OPERAND ADDRESSING MODES

### 5.7.1 Absolute Addressing

The absolute addressing mode is the most common in concept; the data following the machine code is treated as the address of a memory location containing the actual data to be processed during the instruction step. This address is stored in reverse order--as low-byte, then high-byte--to increase processing efficiency during execution

For example:

```
==0230 PIA
       =$4C1F
==0230 LATCH
       =$4DE2
==0230 BUFF1
       =$5400
==0230 START
       =$A000
==0230 EXTRTN
       =$C800
201F4C BIT PIA
C9D7C2 CMP #C2D7
CE8A54 DEC BUFF1+10
4D8054 EOR BUFF1
4C00A0 JMP START
2000C8 JSR EXTRTN
==0242
AD2F6C LDA %11011000
01011111
6ECF11 ROR $11CF
EDCF54 SBC BUFF1+$1F
8DE24D STA LATCH
```

### 5.7.2 Page Zero Addressing

In practice, the zero page addressing mode (identical in concept to absolute addressing) is the most frequently used. This allows the expression of the instruction to be two bytes instead of three; the low byte of the data address is taken from memory, and the high byte is assumed to be zero. All instructions legal in absolute mode are also legal in zero page mode, with the exception of the JMP and JSR instructions (see Table 5-2); the Assembler automatically generates the shortest possible code. It is good programming practice to reserve page zero (memory locations 0-255) for declaration of variables.

NOTE

Any variables on page zero must be defined before they are referenced.

For example:

```
==024E MODE
       =$06
==024E KEY
       =$0C
==024E COUNTR
       =$37
==024E TTYBUF
       =$6B
6537   ADC COUNTR
247A   BIT $7A
C406   CPY MODE
E638   INC COUNTR+1
A66B   LDX TTYBUF
469A   LSR $21+0171
050C   ORA KEY
86AA   STX TTYBUF+$3
F
```



### 5.7.3 Immediate Addressing

The immediate mode of addressing is coded by the character "#" followed by a byte expression; the code inserted into memory is treated as the data to be operated upon according to the machine code.

For example:

```
==025E DOLLAR
       =$24
==025E LAB1
6903   ADC #3
29B5   AND #%10111010
1
E024   CPX #DOLLAR
A945   LDA #'E
A05E   LDY #<LAB1
```

### 5.7.4 Implied Addressing

Twenty-five of the fifty-six instructions, legal only in the implied addressing mode, require no operand--their execution may be completed with no other information than that contained in the opcode. These instructions are preceded by an * in Table 5-2.

For example:

```
00     BRK
D8     CLD
C8     INY
EA     NOP
68     PLA
60     RTS
8A     TXA
```

### 5.7.5 Accumulator Addressing

Instructions implementing the four shift operations have, in addition to addressing modes referencing memory, a special mode which allows manipulation of the accumulator. Usage of this mode, similar to implied addressing, causes generation of a single byte of machine code.

For example:

```
0A      ASL A
4A      LSR A
2A      ROL A
6A      ROR A
```

### 5.7.6 Relative Addressing

Eight conditional branch instructions are available to the programmer; normally these immediately follow load, compare, arithmetic, and shift instructions. Branch instructions uniquely use the relative addressing mode. The branch address is a one-byte positive or negative offset, expressed in twos complement notation, from the run-time program counter. At the time the branch address calculation is made, the program counter points to the first memory location beyond the branch instruction code. Hence, the one-byte offset limits access to branch addresses within 129 bytes forward and 126 bytes backward from the beginning of the branch code. (A one-byte twos complement number is limited to the range -128 to 127 inclusive.) An error will be flagged at assembly time if the branch target lies out-of-bounds for relative addressing.



For Example:

```
9000   BCC *-126
==0275 HERE
F0FE   BEQ HERE
30FC   BMI *-2
707F   BVS *+129
```

### 5.7.7 Indexed Addressing

Indexed addressing (with Index Registers X or Y) facilitates certain types of table processing. The address given as the operand is treated as the base address, to which the contents of either the X or the Y Register is added to arrive at the effective address of the memory location containing the data to be operated upon. All instructions implementing absolute indexed addressing with the X Register also allow the same addressing in the page zero mode; several instructions (LDX, LDY, STX, and STY) allow zero page indexed addressing with the Y Register.

For example:

```
==027B ARRAY
       =$0B
==027B NUMBUF
       =$50
==027B TABLE
       =$2200
794F00 ADC NUMBUF-1,
Y
DDC022 CMP TABLE,X
D60B   DEC ARRAY,X
5DC022 EOR TABLE+$C,
X
B60B   LDX ARRAY,Y
3622   ROL >TABLE,X
968F   STX NUMBUF+$3
F,Y
```

### 5.7.8 Indirect Addressing

The concept of indirect addressing constitutes a level of complexity beyond that of absolute addressing. The operand address references not one memory location containing data, but a sequence of two memory locations containing the address--stored in low-byte, high-byte order--of the loca tion containing the actual data to be processed. True indirect addressing is offered only with the JMP instruction; otherwise, indexed indirect addressing with the X Register and indirect indexed addressing with the Y Register are implemented. For indexed indirect addressing, the indexed address is computed before the indirect is taken; the order of evaluation is reversed for indirect indexed addressing.

NOTE

> Normal indirect addressing takes place when the Index Register contains zero. The JMP indirect uses an absolute-length (two-byte) operand; others require the operand address to lie in page zero between 0 and 254 inclusive.

For example:

```
==028C INDADR
       =$02
==028C CURSOR
       =$57
==028C OLDPTR
       =$7E
==028C NEXT
       =$D9
2102   AND (INDADR,X
)
D17E   CMP (OLDPTR),
Y
6CD900 JMP (NEXT)
B157   LDA (CURSOR),
Y
E102   SBC (INDADR,X
)
8157   STA (CURSOR,X
)
```



## 5.8 ASSEMBLER DIRECTIVES

There are nine assembler directives. They are used to set symbol and location counter values (=), reserve and initial- ize memory locations (.BYTE, .WORD, .DBYTE), and control both assembler input/output (.OPT, .FILE, .END) and assembler listing format (.PAGE, .SKIP). All may be considered as assembly time instructions, rather than as execution-time instructions.

### 5.8.1 Equate Directive

The equate ("=") directive assigns the value of an expres- sion containing no forward references (symbols defined in a following section of code) to either a symbol or the loca- tion counter:

```
==0299
       *=$1800
==1800 TABLE2
       =$C800
==1800 WRDPTR
       =$2A
==1800 NUMPTR
       =WRDPTR+2
```

A label used with an equate directive which increments the location counter will reserve work area memory locations; this is especially useful when consecutively allocating uninitialized memory at the beginning of a program:

```
==1000
        *=0
;EQUATE DIRECTIVE
==0000 EOT
        *=*+1
;RESERVE MEMORY
==0001 EOTADR
        *=*+2
==0003 BUFFER
        *=*+72
==004B ENDFLG
        *=*+1
```

Symbols assigned one-byte values may be programmed as assembler constants--assembly-time values, used consistently throughout a program, which may be changed at a later time when the program is reassembled. Source code is designed so that alteration simply requires reassignment of the corresponding assembler constants. This is considered good programming practice and is a much better alternative to changing each constant as it occurs throughout a program:

```
==004C
        *=0
;EQUATE DIRECTIVE
==0000 STRTCH
        =$28
;ASSEMBLER CONSTANTS
==0000 ENDCH
        =$29
==0000 DELIM
        =$2C
==0000 LOWCH
        =$41
==0000 HIGHCH
        =$5A
==0000 KEYLEN
        =4
==0000 BUFLEN
        =72
```

### 5.8.2 .BYTE Directive

The .BYTE directive initializes byte memory locations. Multiple arguments, separated by commas, may be specified in



a single .BYTE command to load consecutive memory locations; either ASCII strings or expressions evaluating to an eight-bit value are legal. ASCII strings in .BYTE directives must not generate more than 20 characters:

```
==0000 ASCII
4142    .BYT 'ABCD',^
EFH',^'JOE''S'
454648
4A4F
00      ;BYT <ASCII,>
ASCII+2-%1,<*+>*,2
01
0E
02
```

Note the use of two quotes within an ASCII string to insert a single quote into memory.

### 5.8.3 .WORD Directive

The .WORD directive is very useful in constructing jump tables and initializing pointers. An operand expression is evaluated as a two-byte address and is stored in low-byte, high-byte sequence, the order in which the microprocessor fetches addresses from memory. As with .BYTE, multiple operand fields, separated by commas, are allowed:

```
==0010 JMPTBL
04C9    WOR $0804,$E
4B9,$F77A
B9E4
7AF7
0200    WOR $2,<JMPT
BL,>JMPTBL,*,06371
1000
0000
1C00
F90C
```

### 5.8.4 .DBYTE Directive

If it is desired to generate a sixteen-bit expression value in normal high-byte, low-byte order, the .DBYTE assembler directive must be used. Its syntax rules are the same as those for .WORD:

```
==0020 DATA
C804    .DBY $C804,$E
4B9,$F77A
E4B9
F77A
000E    .DBY $E,<DATA
0>DATA,*,%0101101111
01
0020
0000
002C
05BD
```

### 5.8.5 .PAGE Directive

The .PAGE directive causes a title line to be printed under a dashed line. A title may be specified as an ASCII string in the operand field, and it may be cleared with a string of one or more blanks. Absence of an operand will also cause the title to be cleared. This command is not printed as entered in the source code--only the results appear. For example, entry of:



```
.PAGE 'ORIGINAL TITL
E'
.PAGE
.PAGE 'NEW TITLE'
.PAGE
```

would cause the following to appear at the top of each page:

```
--------------------
ORIGINAL TITLE

--------------------

--------------------
NEW TITLE

--------------------
```

### 5.8.6 .SKIP Directive

A blank line may be inserted in the program listing with the .SKIP directive.

```
*=2
 SKI
CURSOR *=*+2
EOT *=*+2
 SKI
TWO =EOT
```

This causes the following listing to be printed:

```
        *=0

==0000 CURSOR
        *=*+2
==0002 EOT
        *=*+2

==0004 TWO
        =EOT
```

### 5.8.7 .OPT Directive

The three options of the .OPT directive control generation of output files and expansion of ASCII strings in .BYTE directives. These options are selected by specifying:

```
.OPT  LIST, GENERATE, ERRORS
```

and are eliminated by coding:

```
.OPT  NOLIST, NOGENERATE, NOERRORS
```

Since only the first three characters of each option are scanned, the following may be written:

```
.OPT  LIS, GEN, ERR
.OPT  NOL, NOG, NOE
```

Of these options, only GEN/NOG remains unspecified after entering Pass 2; GEN/NOG has a default value of NOG.

The three options have the following functions:

1. LIST (NOLIST) controls generation of the program listing, which contains assembled source input, generated object code, errors and warnings.

2. GENERATE (NOGENERATE) controls the printing of object code for ASCII strings in the .BYTE directive. Only code for the first two characters is listed if NOG is specified; otherwise, the whole literal will be expanded.



3. ERRORS (NOERRORS) controls the listing of only erroneous program source lines together with the respective messages generated. Fatal assembler table overflows are also messaged in this file.

### 5.8.8 .FILE Directive

For large programs, it is usually more convenient to divide the source program into logical segments which may be separately loaded into the Editor Text Buffer and edited. After editing, each file is listed from the Text Buffer to a separate file on audio cassette tape. However, when the entire program is to be assembled, it is necessary to tie these files together. This function is performed with the .FILE assembler directive. Each file (except the last) contains, as its last record, a .FILE directive which points to the next file in the chain.

```
.FILE  NAME
```

For example, if the first file is named PRGM, then:

```
.FILE  QARC          is the last statement in file
                     PRGM
.FILE  DEF           IS THE LAST STATEMENT IN FILE
                     QARC
.FILE  PATCH         is the last statement in file
                     DEF
```

### 5.8.9 .END DIRECTIVE

The last statement of the last file in the source program must be the .END directive. For example:

```
.END
```

is the last statement in a one-file program and the last statement of the last file in a multiple-file program.

## 5.9 COMMENTS

Comments may be freely inserted into source code following the last field in a line. If preceded by an opcode (and possibly operand) field, the comment may optionally begin with a semicolon (;). Otherwise, the semicolon is necessary. A comment may be the only field on a line.

For Example:

```
==0000
        *=$0300
==0100
A900    LDA #0
;COMMENT FOLLOWING S
EMI-COLON
A900    LDA #0 COMMEN
T NOT FOLLOWING SEMI
-COLON
```