1 COMPUTER LANGUAGES

In order for a computer to be able to execute a program, the program must first be present in binary form within the computer's memory. However, writing a program directly in binary form is completely out of the question. The solution to this conflict is for the programmer to write a program in a more suitable language and then to use a special computer program to translate it into the binary machine code necessary for the target computer.

When preparing software for a microprocessor system, the designer must first select the particular language in which to write the programs. Three classes of computer languages exist: machine languages, assembly languages, and higher-level languages. Of these, only the last two are normally used by a programmer in preparing programs.

1.1 Machine Language

A computer is a binary machine. The signals used by the various parts of the machine to communicate with each other must be in binary form. A program shown in binary form (or in hex for ease of reading) so as to reflect these values is said to be in machine language. Machine language is the most basic and elementary form in which any program can exist.

1.2 Assembly Language

Before it can be executed a program must reside in memory in machine language.

One step away from using machine language would be to write the program using mnemonics rather than op-codes for the instructions.

A major improvement would be to include some convenient way to keep track of these elements. Assembly language provides this by allowing the user to refer to numerical addresses and constants by name, that is, to use mnemonics for addresses and data as well as for instructions.

Since every program must eventually be in machine language there must always be two versions of each program. The original version prepared by the programmer is called the source program. The machine language version which must be loaded into the computer memory is called the object program.

Figure 1 shows an example of a short segment of an assembly language
program. Part $a$ is the source program and part $b$ is the object program. Observe the word/symbol/mnemonic orientation of the source code as opposed to the strictly binary form of the object code (shown here in hex).

<table>
<thead>
<tr>
<th>(a) source program</th>
<th>(b) object program</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORG 5000H</td>
<td>Address</td>
</tr>
<tr>
<td>LDB #SIZE</td>
<td>5000</td>
</tr>
<tr>
<td>LDX #VEC1</td>
<td>5002</td>
</tr>
<tr>
<td>LDY #VEC2</td>
<td>5005</td>
</tr>
<tr>
<td>LDS #VEC3</td>
<td>5009</td>
</tr>
<tr>
<td>LOOP LDA,X+</td>
<td>500D</td>
</tr>
<tr>
<td>ADDA,Y+</td>
<td>500F</td>
</tr>
<tr>
<td>STA,S+</td>
<td>5011</td>
</tr>
<tr>
<td>DECB</td>
<td>5013</td>
</tr>
<tr>
<td>BNE LOOP</td>
<td>5014</td>
</tr>
<tr>
<td>SWI</td>
<td>5016</td>
</tr>
<tr>
<td>FCB 8</td>
<td>5017</td>
</tr>
<tr>
<td>SIZE EQU 20</td>
<td></td>
</tr>
<tr>
<td>VEC1 EQU 1000H</td>
<td></td>
</tr>
<tr>
<td>VEC2 EQU 2000H</td>
<td></td>
</tr>
<tr>
<td>VEC3 EQU 3000H</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C614</td>
</tr>
<tr>
<td></td>
<td>8E1000</td>
</tr>
<tr>
<td></td>
<td>108E2000</td>
</tr>
<tr>
<td></td>
<td>10CE3000</td>
</tr>
<tr>
<td></td>
<td>A680</td>
</tr>
<tr>
<td></td>
<td>ABAO</td>
</tr>
<tr>
<td></td>
<td>A7EO</td>
</tr>
<tr>
<td></td>
<td>5A</td>
</tr>
<tr>
<td></td>
<td>26F7</td>
</tr>
<tr>
<td></td>
<td>3F</td>
</tr>
<tr>
<td></td>
<td>08</td>
</tr>
</tbody>
</table>

**FIGURE 1**

The conversion from the source program into the object program is the sort of a program, which translates the source into the machine code object program. If the source program is written in some higher-level language such as FORTRAN or Pascal then the translation program is called a *compiler*. If it is written in assembly language then the translation program is called an *assembler*.

1.3 Assembly Language versus Higher-level Languages
Most programmers are capable of writing a relatively constant number of program statements per day regardless of the programming language used. An assembly language program requires on the order of ten times as many statements as an equivalent higher-level language program. However, assembly language may be a suitable choice in some cases.

A good assembly language: (i) occupies less space in memory, (ii) generally run faster than their higher-level counterparts.

Even in applications where a higher-level language is used for the bulk of the
system programs, certain segments of the code may be written in assembly language. In many simple control applications it is not unusual for all of the programming to be done in assembly language.

1.4 Characteristics of Assembly Languages

Although assembly languages are processor-dependent they do have some common characteristics. All assembly languages use mnemonics to specify the processor instructions and certain supporting operations required of the assembler itself.

All assembly languages include a set of directives, which are used by the programmer to issue commands to the assembler itself. These are called pseudo-instructions since they resemble instructions in format although they are not instructions. They are also known as assembler directives, since they are commands or directives issued to the assembler. For example, the origin directive, whose mnemonic is (universally) ORG, may be used to specify the origin or starting address of a section of the program.

All assemblers provide for the use of symbolic labels which may be affixed to constants and memory addresses. These labels may be used as substitutes for the numbers with which they are identified. The use of labels rather than absolute numbers is a powerful programming tool.

A branch target address, too, is easier to implement in the program when the programmer need not be concerned with its actual value or its distance but may refer to it by name.

Constants may be specified in one of several different bases with decimal, hexadecimal, octal, and binary being common options.

The set of rules regarding the proper form for statements in the assembly language is called the syntax of the assembler. The syntax for most assemblers divides each statement into four fields: from left to right, the label, mnemonic or operation, operand, and comment fields.

An assembler scans the source program twice while translating it into machine code. During the first pass it counts bytes and locates all of the label definitions, which it places into a table along with the corresponding numerical values. During the second pass it uses this label table or symbol table to generate the machine code. This feature is particularly important in branches where the target address may be defined by being a label on a later instruction.
2.1 The Character Set
The following characters are recognized by the assemblers:

1. The alphabet A through Z
2. The integers 0 through 9
3. Four arithmetic operators: + — * / 
4. Characters used as special prefixes:
   a. # specifies the immediate addressing mode
   b. $ specifies a hexadecimal number
   c. % specifies a binary number
   d. @ specifies an octal number
5. Characters used as delimiters:
   a. * or ; in column 1 specifies a comment line
   b. * or ; or space separates comment from operand
   c. a space separates operation from label and operand from operation; this is also the standard delimiter between operand and comment in Motorola assemblers
   d. Comma separates multiple operands
6. Characters with special meanings in certain places:
   a. * in operand stands for present location in memory
   b. " in operand stands for beginning and end of a string of ASCII characters
   c. Letters used as suffixes in numbers:
      i. B denotes a binary number
      ii. D denotes a decimal number
      iii. H denotes a hexadecimal number
      iv. O or Q denotes an octal number
7. A comment may include any printable character.

2.2 Program Statements
A source program is composed of a sequence of statements, one per line. These statements must be written into an ASCII file with an editor program.

A source statement includes between one and four fields. From left to right these are:

(1) label (2) operation (3) operand (4) comment

A label is required with some statements involved in the definitions of symbols; otherwise it is optional. The use of labels on branch destinations, although optional, will greatly reduce the programming effort and opportunity for error. The
operation or mnemonic must be included in every statement except those, which are purely comment. An operand may or may not be required, depending upon the nature of the operation. The comment is always optional and may be included in any statement. Comments are intended for the convenience of the programmer and to facilitate proper documentation of the program.

The standard delimiter or field separator in Motorola assemblers is the space, however also allow the use of the asterisk or the semicolon to separate the operand from a comment.

```
ORG 1000H  6809 Program Starts In Location 1000
START LDA FIRST  Pick Up First Operand
ADDA SCND  Add Second Operand
STA SUM  Store The Result
SWI  Return To Monitor
FCB 8  Monitor Code
BRA START  Start Again
FRST FCB 34H  Data Area
SCND FCB 56H
SUM FCB 0
END
```

### 2.3 The Label Field

Labels always start in the first column of the statement line. If a label is not used in a particular statement, that statement must begin with a space. Except when it is used with the assembler directives EQU or SET (described below) to define a constant, a label corresponds to a numerical address in the computer. The label provides a convenient means for the programmer to refer to that address within the operand field of other statements in the program. The programmer need only use the identical symbols.

The following rules apply to labels:

1. A label consists of one or more alphanumerical characters. ASSYM09 will accept a maximum of six characters, ASSYM000 will accept a maximum of 16 characters.
2. The first character of a label must be alphabetic.
3. A label must begin in the first column of the statement.
4. Each label used in a program must be unique.

Although the above rules are quite unrestrictive, it is good practice to follow the following "unwritten rules" in devising labels:
1. Use labels which are meaningful in the context of the program, i.e., SPEED rather than X7.
2. Avoid using a label, which duplicates any mnemonic of the assembly language. Some assemblers specifically exclude these as labels and their use can be confusing.
3. Do not use labels which are too similar to each other. Although the assembler can distinguish between PARTI and PART1 or between IIIII and IIIII, a human reader will have difficulty.

2.4 The Operation Mnemonic Field
The operation mnemonics recognized by an assembler include the executable instructions of the respective processor and a set of assembler directives. Each instruction mnemonic, together with its addressing mode and operands, is translated by the assembler into the appropriate machine code. Certain of the assembler directives result directly in the generation of binary code as part of the object program. Others simply control the assembly process and do not call for any binary code.

2.5 Assembler Directives
The following list describes some of the more common Motorola assembler directives and includes an example of each.

**ORG** Defines ORIGIN of the code. Tells the assembler where in memory to start assembling code.

```
ORG      1000H
```

*sets the starting address of the subsequent code to be 1000H*

**EQU** Permanently defines the value of the label to be EQUAL to the value of the operand. A label is required.

```
CR     EQU     ODH
```

*makes the label "CR" equal ODH*

**EXTERN** Imports the value of a label from another module so that the current module may reference it. The other module must also declare the label(s) as global (see GLB).

```
EXTERN TIME,SIZE
```

*import the values of TIME and SIZE from another module*

**GLB** Makes a label available to be exported to another module, which will be linked with the current one before they are executed.

```
GLB  TIME,SIZE
```

*declares that the labels TIME AND SIZE may be used in another module and that their values are defined in the current module*

**END** The last statement in the source program. It tells the assembler to start the assembly process.

**ABS-SHORT** Changes the default absolute addressing mode in the ABS_LONG

MC68000 to either short (16 bits) or long (32 bits).
**DC.X** Define Constant. Stores the operand list in memory using one byte per operand if \(X\) is B, two bytes per operand if \(X\) is W, and four bytes per operand if \(X\) is L. May have a label. Operands may be expressed as an ASCII string enclosed within quotes.

\[\text{DC.B} \quad 4C,100,'\text{Do it.}'\text{ stores the hex values } 4C, 64, 44, 6F, 20, 69, 74, 2E \text{ in memory starting at the current address}\]

**DS.X** Define Storage. Reserves memory locations without placing any values in them.

\[\text{ARRAY DS.L 15 reserves 60 consecutive bytes (15 long words of four bytes each) starting at the current address which will be named ARRAY}\]

**EVEN** Adjusts the assembler's location counter to the next even address if it is not already even.

### 2.6 The Operand Field

The form of the operand will vary depending upon the operation mnemonic in the statement. The operand may convey addressing mode information; it may list data to be stored in memory locations by the assembler; it may provide an address for an EQU or an ORG directive; or it may not be present because it is neither required nor allowed.

When any part of the operand conveys numerical information, it may be included in any one of several alternative ways. Numbers may be included directly in one of the supported number systems; they may be expressed in terms of the location of the current instruction plus an offset; they may be instruction labels or other address or constant labels; or they may be arithmetic expressions involving any of these. The following examples illustrate some of the possibilities:

#### Direct Numerical

a) Decimal: 1234 or 1234D Decimal is the default number system. Numbers with no other prefix or suffix are always interpreted as decimal.

b) Hex: $1234$ or 1234H Numbers must start with digits 0-9. A234H is invalid; it is interpreted as a label! Use $A234$ or 0A234H.

c) Octal: 1234Q or @1234 or 1234O (letter O not digit 0)

d) Binary: 01001001B or %01001001

#### Offset from Current Location

*+15* This number is the address of the current instruction op-word plus 15.

*%-35* This number is the address of the current instruction op-code minus 35.

#### Constant Labels

HNDRD EQU 100 Defines the label HNDRD as 64 hex.
LDA #HNDRD     The immediate operand in the instruction is evaluated as 64 hex.

**Address Labels**
ORG $567A     Defines the origin.
CMD FCC "Do it!" Defines the label CMD as the value 567AH.
CLR CMD,Y     The operand is evaluated as $567A,Y.

**Arithmetic Expressions** (using defined labels from above)
LDA #2*HNDRD  Coded as LDA #200 or LDA #$C8
LDX CMD-$0A   Coded as LDX $5670

2.7 The Comment Field
Comments constitute an important part of the documentation of the program and serve to assist a reader in understanding the program. They are completely ignored by the assembler except that they are included in the program listing.