ITHACA INTERSYSTEMS
ASMBLE/Z
A RELOCATING MACRO ASSEMBLER
REVISION 2.0

Copyright by
© Ithaca Intersystems, Inc.

ASMBLE/Z

A Relocating Macro Assembler

© Copyright 1980 by
Ithaca InterSystems, Inc.

Manual Revision 2

TABLE OF CONTENTS

INTRODUCTION

ASMBLE is a Z-80 source code macro assembler which produces either an absolute binary, a hex, or a relocatable code module and a program listing. The assembler allows you to specify the devices and file names for the input and output files as well as which output files you want generated. If you ask for a listing, it will contain a column alphabetized symbol table.

FEATURES

1) Two pass operation

2) Conditional assembly

3) External labels and relocatable code

4) Absolute binary or hex code

5) Separation of code and data spaces

6) Macros

7) Include files

8) Column alphabetized symbol table in listing

WHO SHOULD USE THIS MANUAL?

You may be reading this manual because you want to know how to assemble, link, and run a program written in Pascal. If this is the case, you should skip this manual and read the first part of the linker manual since the Pascal compiler takes care of generating all the assembler code that is normally required to run a Pascal program.

On the other hand, you may want to add your own assembler routines to a Pascal program, or you may want to write a stand-alone assembler program. In that case, you should read this entire manual and then read the linker manual.

ASSEMBLING

Here are the steps you go through from the creation of an assembly language program to its execution.

1) Run the text editor, create a new SRC file on a disk, and type in your program (written in assembler mnemonics).

2) Run the assembler which translates the assembler mnemonic statements into machine language code.

3) If you asked the assembler to generate a relocatable module, you run the linker which loads the module into memory, and start the program.

   If you asked the assembler to generate hex code, you run the loader which translates the hex code into a COM file. You then run the COM file.

   If you asked the assembler to generate a COM file, you run it.

In your program you tell the machine exactly what to do by writing a list of mnemonic machine instructions. These mnemonics are translated, one to one, into machine executable instructions (machine code).

Each machine instruction has its corresponding mnemonic. For example, if you want to move a copy of the byte from the B register into the A register you write:

```
MOV     A,B
```

If you want to complement the byte in the A register you write:

```
CMA
```

An instruction is one or more bytes long and is stored in one or more consecutive memory locations. You can symbolically reference an instruction by placing a label in front of the instruction and referencing the label. For example, suppose you want to write a routine that decides whether or not a value in the A register is equal to ten. You might write:

```
        CPI     10      ; Does A = 10?
        JZ      TARGET  ; Yes.
        XRA     A       ; No.  Make it 0.
TARGET:MOV      B,A     ; Save it in B.
```

In this example, if the A register contains ten, the machine jumps to the instruction bearing the TARGET label. You can locate this routine anywhere in memory and not worry about the location which TARGET represents (the value of TARGET). The assembler calculates it for you.

You may also give a value to a symbol with an equate instruction.

```
ENDVAL EQU      10
```

Here, ENDVAL is given the absolute value ten. It has this value no matter where it is defined. You may use it in your program as follows:

```
ENDVAL EQU      10
       CPI      ENDVAL  ; Does A = ENDVAL?
       JZ       TARGET  ; Yes.
       XRA      A       ; No.  Make it 0.
TARGET:MOV      B,A     ; Save it in B.
```

A section of an assembly code may be switched on or off by surrounding the code with conditional statements. For example:

```
FALSE   EQU     0
TRUE    EQU     NOT FALSE
ORANGE  EQU     TRUE
        IF      ORANGE
NAME:   DB      'This program is called Orange'
        ELSE
NAME:   DB      'This program is called Lemon'
        ENDIF
```

In this example, the first three statements define the values of the symbols FALSE, TRUE, and ORANGE. The conditional statements, IF, ELSE, and ENDIF, select one of the two statements labeled NAME for assembly.

Code which is often repeated, possibly with some variation, may be stored in a macro and assembled simply by giving the name of the macro. This saves a little typing and usually makes the program easier to understand. For example:

```
PRINT: MACRO   TEXT
       LXI     H,TEXT
       CALL    TEXT
       ENDMAC
         .
       PRINT   HITEXT
         .
         .
HITEXT:DB      'Hi there'
```

In this example, the first four lines define the macro called PRINT which contains one dummy parameter, TEXT. When the statement PRINT HITEXT is assembled it is replaced by:

```
       LXI     H,HITEXT
       CALL    TXTYP
```

This loads the address of the text string into the HL register and calls TXTYP to print the string.

Here is a short example of a program that reads the front panel switches and sets the front panel lights accordingly. It contains a conditional control switch, FLIP, which causes the assembler to generate code to complement the value sent to the lights if FLIP is true. If all the switches are up the program returns control to the monitor.

```
; Light Test.

FALSE   EQU     0
TRUE    EQU     NOT FALSE

FLIP    EQU     TRUE    ; Complement flag.

LIGHTS  EQU     0FFH    ; Front panel lights port.
SWITCH  EQU     0FFH    ; Front panel switch port.

ENDCOD  EQU     11111111B; Switch pattern for stop.

NDTEST: MACRO           ; Test for ENDCOD in A.
        CPI     ENDCOD  ; Time to quit?
        JZ      0       ; Yes.  Back to monitorland.
        ENDMAC

        ORG     100H    ; Put the code at location 100H.

LOOP:   IN      SWITCH  ; Read the switches.
        NDTEST          ; Test for end.

        IF      FLIP
        CMA             ; Flip the bits.
        ENDIF

        OUT     LIGHTS  ; Display in the lights.
        JR      LOOP    ; And repeat.

        END     LOOP    ; Start at LOOP.
```

Here is a listing of the program in the previous example as it is assembled. Notice how the macro and conditional code is treated.

```
Light Test.                                          ASMBLE v-5b Page 1

                        ; Light Test.

     0000       FALSE   EQU     0
     FFFF       TRUE    EQU     NOT FALSE

     FFFF       FLIP    EQU     TRUE    ; Complement flag.

     00FF       LIGHTS  EQU     0FFH    ; Front panel lights port.
     00FF       SWITCH  EQU     0FFH    ; Front panel switch port.

     00FF       ENDCOD  EQU     11111111B; Switch pattern for stop.

     0000       NDTEST:MACRO            ; Test for ENDCOD in A.
    -                   CPI     ENDCOD  ; Time to quit?
    -                   JZ      0       ; Yes.  Back to monitorland.
                        ENDMAC

     0100               ORG     100H    ; Put the code at location 100H

0100 DB FF      LOOP:   IN      SWITCH  ; Read the switches.
                        NDTEST          ; Test for end.
0102+FE FF                CPI     ENDCOD  ; Time to quit?
0104+CA 0000              JZ      0       ; Yes.  Back to monitorland.

     FFFF               IF      FLIP
0107 2F                 CMA             ; Flip the bits.
                        ENDIF

0108 D3 FF              OUT     LIGHTS  ; Display in the lights.
010A 18 F4              JR      LOOP    ; And repeat.

     0100               END     LOOP    ; Start at LOOP.
```

Here is a listing of the same program except that FLIP has been set to false. Notice how this changes the conditional code.

```
Light Test.                                          ASMBLE v-5b Pa  1

                ; Light Test.

     0000       FALSE   EQU     0
     FFFF       TRUE    EQU     NOT FALSE

     0000       FLIP    EQU     FALSE   ; Complement flag.

     00FF       LIGHTS  EQU     0FFH    ; Front panel lights port.
     00FF       SWITCH  EQU     0FFH    ; Front panel switch port.

     00FF       ENDCOD  EQU     11111111B; Switch pattern for stop.

     0000       NDTEST:MACRO            ; Test for ENDCOD in A.
     -                  CPI     ENDCOD  ; Time to quit?
     -                  JZ      0       ; Yes.  Back to monitorland.
                ENDMAC

     0100               ORG     100H    ; Put the code at location 100H.

0100 DB FF      LOOP:   IN      SWITCH  ; Read the switches.
                        NDTEST          ; Test for end.
0102+FE FF                 CPI     ENDCOD  ; Time to quit?
0104+CA 0000               JZ      0       ; Yes.  Back to monitorl  .

     0000               IF      FLIP
                        CMA             ; Flip the bits.
                        ENDIF

0107 D3 FF              OUT     LIGHTS  ; Display in the lights.
0109 18 F5              JR      LOOP    ; And repeat.

     0100               END     LOOP    ; Start at LOOP.
```

ASSEMBLER OPERATION

The assembler operates in two passes. Its operation is almost identical in both passes.

During the first pass the input file is read and each source line is processed. Each time a symbol is defined it is entered into the symbol table. All error messages except SYMBOL NOT FOUND, REDEFINED, and OUT OF RANGE are printed during pass one.

During pass two the input file is read again and each source line is processed. If the source line generates any machine code, it is sent to either the binary, hex, or relocatable output file in the proper format. A copy of the line of source text along with the address and generated machine code is sent to the listing file.

ONCE THRU CODE

The initialization routine used by the assembler is written in once through code and is located in the symbol table region. You may restart or save the assembler at any time while the assembler is asking the file name question. After the question has been answered, the assembler no longer needs the initialization code and destroys it. From this point on the assembler must be reloaded in order to restart it.

There are two terms, module and section, which have special meanings when used to describe relocatable code. When one or more source files are assembled the resulting relocatable code is called a module. That is, each time the assembler is used to generate relocatable code it produces a single module. The module may contain one or more sections. It may contain a program section, a data section, and one or more common sections.

ASMBLE/Z can produce relocatable modules. These modules are loaded into memory by the linker. Taken together, these two programs (the assembler and the linker) provide very powerful facilities for the programmer:

1) Source code (in a SRC file), written with all address references represented by symbols, can be assembled into a relocatable module, which is then sent to the linker. The linker can be told to load the module at nearly any address; that is, the module is relocatable. The relocatable modules require relatively little processing by the linker as compared to the processing the assembler performs on a source file, and therefore the relocation of a module can be accomplished in very little time.

2) Several relocatable modules can be loaded by the linker into different locations in memory; the linker determines the absolute addresses so that all the code is loaded properly, each relocatable module going into the next memory location left free by the last relocatable module.

3) Convenient means are provided to allow various relocatable modules to make references to each other. This means that there can be symbols in a source file which only reference other places in the same source file (this means that they can be duplicated in other source files you wish to link without conflict) and, on the other hand, certain labels can be specified as entry point symbols or external symbols (see ENTRY and EXTERNAL section), allowing different source files to have common symbolic references. If in one source file a certain symbol is specified as an entry point, then references to that symbol in other modules - if they are declared as external symbols - will be performed correctly. Thus, modules can call subroutines and reference data in other modules.

4) Also available are common sections. These are typically used to transfer data between different modules. Each relocatable module may have up to 15 of these, distinguished

by their name - any symbolic label desired, or a blank label, is permitted. When the various source files are assembled into relocatable modules and are then loaded into memory by the linker, these common sections are grouped together from all the different modules according to name. The common sections are overlayed; the linker assumes that any common sections with the same name (all blank commons are assumed to have the same name) represent identical locations in memory. This allows the different modules to have common tables of data, so that when one module calls a subroutine in another, for instance, it can pass a reference to a table in a common area that the other subroutine can use to process data.

5) Other named sections are provided: PROG, ABS, and DATA. The PROG (program) is the default section which is assumed if no section label is given. ABS provides the facility for writing absolute code that will not be relocated, when that is desired. DATA is provided so that you may, if you desire, locate the data section of a program in a different area than the instruction area, as might be necessary if the program is to be burned into a PROM.

6) A facility of the linker that provides even more programming power is the ability to construct library modules. These are produced in much the same way as a normal relocatable modules - by writing source files, assembling them to produce relocatable modules, and linking them - except that in the librarian mode, the linker produces a library file as output instead of executable absolute binary code. This library file contains relocatable modules, but provides a powerful additional feature. You typically load one or more relocatable modules with the linker, followed by a library file; the linker treats the library file in a special way, in that, as it encounters each module in the library file, it checks to see if any references have been made to the entry point symbols in that library module. If the linker finds no requests for these entry points, it skips that module of the library file without loading it, and moves on to the next. On the other hand, if the linker finds that it needs one or more entry points in the library file it loads that module.

These features together provide a very flexible Z-80 assembly language environment. The operation of each facility is explained in detail in later sections of this manual.

MODULE SECTIONS

A relocatable module may contain up to eighteen different sections to allow you to store absolute, program, data, and common code. Each section has its own program counter. At the beginning of an assembly all program counters are set to zero. As the assembler generates code in one section the appropriate program counter is incremented to keep track of the location of each byte of code. As labels are generated they are marked as belonging to that section of code. When you change from one section to another (you may do this as often as you like) the assembler saves the program counter from the last section and loads the program counter for the new section. Later on, if you switch back to the previous section again the program counter points to the next available byte in that section and the code assembly continues from where it left off.

For example, if you generate three bytes in section one they are stored at locations 00, 01, and 02 in section one's base. Then you generate two bytes in section two. They are stored at locations 00 and 01 in section two's base. Now, if you generate another byte in section one it is stored at location 03 in section one's base.

The eighteen different sections are called by name. The first three sections are called ABS, PROG, and DATA (when the assembler starts it specifies PROG as the default section). The remaining fifteen sections are called COM. Each COM section has a user defined name. The names are only significant in the first eight characters (the same as symbol names). One COM section may be unnamed. It is referred to as blank common. You may change to any section by giving its name as an instruction. For example:

```
DATA            Start the DATA section.
PROG            Start the PROG section.
COM     TABLE   Start a common section named TABLE.
COM             Start a blank common section.
DATA            Continue in the DATA section.
```

The assembler treats all eighteen sections alike. That is, it maintains a separate program counter for each section and marks all labels generated in a given section as belonging to that section. The linker, on the other hand, treats the sections differently. Absolute code from the ABS section is always loaded into absolute memory as specified (that is, it is not relocated). If several modules are being linked together the PROG and DATA sections from the various modules are loaded into different regions of memory. All common sections of a given name are loaded into the same locations. For example, assume the

linker loads two modules which each contain a PROG section and a blank common section. When the program runs, the part of the program in the first module's PROG section might store a data byte in the first location of blank common. The part of the program in the second module's PROG section might load the same byte from the first location of blank common.

You should use a little caution when generating code in ABS and COM sections. This code may be overwritten by other modules which are linked together. For example, one module may initialize a table in a common section in one way and another module may initialize the table in the same common section in another way. The order in which the modules are specified to the linker determines which initialization is overwritten and which one remains loaded. It is usually better to simply reserve space in all common sections with the DS instruction and initialize them at run time.

You may also change sections with an ORG instruction. The type of argument (that is, the section in which the argument was defined) determines the new section. For example, if BLOTZ is the name of a location in the data section, then:

```
        ORG     BLOTZ+27
```

tells the assembler to generate code in the data section 27 bytes beyond BLOTZ.

You should be very careful about using the ORG instruction in programs that use external symbols (see ENTRY and EXTERNAL section, below). The assembler generates all references to an external symbol of a given name as a linked list. The last reference points to the previous reference, etc. The list must be intact for the linker to properly resolve the external symbol. If you rewrite a section of code with the ORG instruction (for example, ORG $-20) and an external reference is overwritten, then the linked list is broken and the linker will do unpredictable things. This cannot happen if you use the PROG, and DATA instructions.

Referencing external symbols in a common section is also a dangerous practice because the linker overlays all common sections of the same name. In general, it is not a good practice to store any executable code in common sections.

ENTRY and EXTERNAL

Modules may communicate with each other through common sections as explained in the previous paragraphs. They may also communicate by specifying various locations as entry points in one module, and as an external symbol in another module. The linker matches up all the entry point symbols in one module with all the external symbols in other modules it is linking.

Entry points and external symbols are treated as sixteen-bit address values. Therefore, if BLOTZ is an external symbol you may refer to it in a statement such as LXI H,BLOTZ, but you may not refer to half of an external address in a statement such as MVI A,BLOTZ/256.

A module may specify certain locations as entry points, in which case they must be defined in the same module. The module may also specify certain locations as external to that module. These locations must not be defined in that module but will be defined later in the linking operation. For example, suppose you are writing a navigation module which uses trig functions (subroutines) in another module. In your navigation module you might write:

```
        EXT     SIN,COS,TAN     ; define SIN, COS, TAN as externals
        LHLD    ANGLE
        CALL    SIN
          .
          .
        LHLD    ANGLE
        CALL    COS
          .
          .
```

In the trig function module you might write:

```
        ENTRY   SIN,COS,TAN     ; define SIN, COS, TAN as entry points.
SIN:    PUSH    H
          .
          .
        POP     H
        RET
COS:    PUSH    H
          .
          .
```

When the linker links these modules it first loads them into memory and determines the actual locations of the three entry points, SIN, COS, and TAN, in the trig function module. Then it goes through the navigation module and sets the actual addresses for the three external symbols.

NAME

Every relocatable module has a name. The name is initially set to the first eight characters of the output file (REL file) name. You may change the module name at any time with the NAME instruction. You may change the name as often as you like but only the last name specified is given to the output file. For example:

```
NAME    TRIG
```

Related relocatable modules, usually subroutines, may be collected together in a single file called a library. Various modules from the library are selectively loaded by the linker after the main routines (modules) of a program are loaded. That is, the main routines of a program usually need to use subroutines which are found in the library. The main routines are loaded first. Whenever a main routine needs a library subroutine it declares the subroutine's entry point to be external to the main routine. The linker places the subroutine entry point name (symbol) in the external symbol table. After the main routines have been loaded (and several symbols have been placed into the external symbol table) the linker selectively loads the library. It compares entry point symbols from each library module with symbols in the external symbol table. If it finds a match, that is, if it finds that one or more entry points in a library module will resolve external symbols, it loads the module. If it does not find a match it skips the module (since it does not need it) and goes on to the next one.

A module in a library may contain external symbols as well as entry point symbols, that is, the module may require the services of one or more other modules in the library. For example, in the TRIG library, the TAN and COT modules calculate the tangent and cotangent of an angle. These modules make use of the identity: TAN(a)=SIN(a)/COS(a), and call the SIN and COS modules to calculate the sine and cosine of an angle. The TAN and COT modules also call the DIV module to perform the division.

A library should load all necessary modules (and no unnecessary ones) in one pass. This means that a module should appear in a library after it has been referenced by an external symbol in other modules. That is, external symbols should forward reference the modules in which the symbols are defined (as entry points). For example, the COT module should come before the TAN module because it uses the TAN function in its calculation (COT(a)=1/TAN(a)). The TAN module should come before the SIN and COS modules. The SIN and COS modules do not reference each other and thus may appear in any order. Everything references the DIV module so it should come last. With the library put together in this order the required modules (but no more) will be loaded no matter what the main routine may require.

Sometimes it is not possible to arrange library modules so that their external symbols only reference in a forward direction. For example, suppose that some subroutines in module A reference some subroutines in module B and some other subroutines in module

B reference some subroutines in module A (fold your hands and think about it). There are several things you can do to rectify this situation: You may decide that module A and module B should be combined into one larger module thus eliminating the cross referencing. Or you may find that you can eliminate the cross referencing by moving some subroutines from one module to the other.

However, it may not always seem possible to eliminate the cross referencing. In that case you may put two copies of module A into the library, one before and one after module B. If the main routine needs module A it is loaded the first time it is encountered in the library. Module A then references module B which is loaded next. When the second copy of module A is encountered in the library it is skipped because all external references to it have already been resolved (resolved external symbols are removed from the external symbol table). On the other hand, if the main routine needs module B it is loaded first followed by module A. In either case both modules are loaded, only their order in memory is different.

It is a good idea to put non-modifiable execution code (pure code) in PROG sections and modifiable data in DATA sections. This is true in both main routines and in libraries. If you ever want to burn a program into a PROM you simply tell the linker to load all PROG sections into the PROM region of memory and to allocate space in read-write memory for the data. For example, suppose you have a pair of text buffering subroutines: one subroutine gets a complete line of text from the keyboard and puts it into a line buffer, the companion routine returns the next sequential character from the line buffer each time it is called. These two subroutines would be placed in the PROG section of the module and the line buffer would be placed in the DATA section.

The assembler evaluates each line of source code and generates one or more bytes of machine code. The machine code will be loaded into sequential memory locations later on. The assembler keeps track of the current memory address in its program counter. This is a 16 bit counter which starts with a value of zero. Each time the assembler generates a byte of machine code, it increments the program counter. Since each byte is stored in a location whose address is one greater than the address of the last byte, the value of the program counter and the value of the current memory address always agree. This one-to-one correspondence is, of course, altered when a relocatable module is loaded by the linker.

The program counter may be read with the symbol $. In the following examples | represents the left edge of the source line.

```
|HERE   EQU     $       HERE is set to the current value of the
                        program counter.
```

There are actually eighteen different program counters; one each for the absolute, program, and data sections, and one for each of the fifteen different common sections. Every time a new section is entered the program counter for the last section is saved and the program counter for the new section is loaded. This means that you can generate code in a program section, for example, then switch to the data section, generate some data code, then switch back to the program section and continue generating code from where you left off.

SYMBOLS

A symbol represents a number or an instruction. It starts with a letter, dollar sign, percent sign, dot, number sign, or underscore and may contain any of the following characters:

0-9	Numbers
A-Z	Upper case letters
a-z	Lower case letters
$	Dollar sign
%	Percent sign
.	Dot
#	Number sign
_	Underscore

Here are some examples of symbols and non symbols:

$	A symbol may start with $.
ABC	A symbol may start with a letter.
X27	A symbol may contain numbers.
4SALE	A symbol must not start with a number.
D^3	A symbol must contain only alphanumeric characters, $, %, ., #, _.

When a symbol is evaluated all lower case characters are translated into upper case characters. The following symbols all have the same value:

```
mov
Mov
MOV
```

When the assembler extracts a symbol from a source line, it picks up characters until it has a total of eight characters or until it reads a non-symbol character. Any symbol characters beyond the first eight are ignored. Here is a list of symbols as they appear in a source line and as they are extracted by the assembler:

abc123	ABC123
A,B	A
VALUE12	VALUE1
VALUE13	VALUE1

In the first example the lower case characters are translated into upper case characters. In the second example the symbol is A and is terminated by the comma. In the third and fourth examples only the first eight characters are significant in the symbol. The rest are ignored. Notice that VALUE12 and VALUE1

are treated as the same symbol.

LISTING

The first line on each page of the listing is the program header line. It is made up of the first line from the first source file (with leading semicolons, spaces, and tabs stripped off), the current date, the assembler version number, and the current page. The remainder of the page contains the program listing.

Each listing line contains the address of the first byte of code in the line, up to four bytes of code, and the source text which generated the code.

The DB and DS instructions may generate more than four bytes of code. In this case the extra code is listed on subsequent lines.

Some instructions do not generate any executable code (for example, EQU, IF, END, etc). The address is left blank in these lines to indicate that no code is generated. However, many of these instructions have a numeric value associated with them which is listed.

Addresses associated with relocatable (non absolute) code sections are followed by various characters to indicte the code section in which they were generated. The characters are as follows:

'	PROG
"	DATA
*	COM
#	EXT

Macro definitions are noted with a minus sign following the address. Macro expansions are noted with a plus sign following the address.

Macro definitions, macro expansions, and conditional statements (IF) may be nested (a macro expansion may call another macro expansion, for example). The source text is indented two spaces for each level of nesting.

Sixteen-bit values are listed with their most significant byte first for readability, but they are stored with their least significant byte first. For example, the following instruction

```
LXI     B,1234H
```

is listed as:

```
0000 01 1234    LXI     B,1234H
```

and generates the following code:

```
01
34
12
```

The symbol table follows the program listing. The first line contains information about the assembly (number of errors detected, number of symbols generated, and amount of unused space in memory). If the program generated any macros the next line contains information about the macros (number of characters stored and number of macros generated). The next line contains information about section sizes (size of absolute, program, and data sections) followed on subsequent lines by the names of all common sections and their sizes (blank common is listed as * *). The symbols follow on the next page in column alphabetized order followed by their sixteen bit value written as four hex characters. If a value is a relocatable address it is followed by the corresponding relocation character (', ", *, or #). Macro names are also listed in the table followed by the letter M in place of the value.

SOURCE LINE FORMAT

A source line consists of a label field, an instruction field, an argument field, and a comment field. Each line may contain none, any, or all of these fields. This is what a source line looks like:

```
|LABEL   INSTRUCTION   ARGUMENT(S)      COMMENT
```

LABEL FIELD

A label is a symbol which begins in the first column. If a symbol does not begin in the first column it is not a label. This means that you may have only one label on a line since there is only one first column on a line. It also means that you may not indent labels.

```
|BOB            BOB is a label.
| CHARLIE       CHARLIE is not a label; it is indented.
```

A label may be terminated with any non symbol character, that is, a space, tab, colon, etc.

```
|MULT           Label ends with a space.
|DIV:           Label ends with a colon.
```

The symbol used in a label is given the current value of the program counter. Since the value of the program counter is equivalent to the current memory address, each label is equal to the memory address of the first byte in its line. For example, suppose that the current value of the program counter is 123.

```
|MIX:  MOV      A,B
|MATCH:MOV      C,D
```

MIX is given the value 123 since the value of the program counter is equal to 123 at the beginning of the first line. The instruction MOV A,B generates one byte of code. This increments the program counter. At the beginning of the second line it has a value of 124 so MATCH is given a value of 124. In the case of relocatable code, the assigning of actual memory addresses to labels is deferred until the linker loads the code.

INSTRUCTION FIELD

An instruction is a symbol which does not begin in the first column. The assembler tells the difference between labels and instructions by noting whether or not the symbol starts in the first column. The instruction symbol may only be terminated with a space, tab, semicolon, or carriage return.

```
|TOP:  RAL      TOP is a label.  RAL is an instruction.
| PCHL         PCHL is an instruction.
|              It does not start in the first column.
|L26:CMA       L26 is a label terminated by a colon.
|              CMA is an instruction.
```

ARGUMENT FIELD

Some instructions require one or more arguments. The arguments are separated from the instruction by one or more tabs or spaces. If the instruction requires more than one argument the multiple arguments must be connected by commas and must have no intervening tabs or spaces. The only exception to this rule is the use of the arithmetic operator NOT. It must be separated from the argument it modifies by a tab or a space. Here are some examples of single arguments:

```
COUNT          A symbol
C              Either the symbol C or register C
'G'            A one byte text string
'AB'           A two byte text string
'Time'         A multi-byte text string
36             A number
NOT TRUE       An arithmetically modified symbol
TOP+2          Another arithmetically modified symbol
```

Here are some examples of instructions which require single and multiple arguments:

```
|       POP     D
|       ADI     100
|       SUI     PVAL

|       MOV     C,A
|       LXI     H,ADDR
|       LXI     B,'XY'
```

In the first example the instruction POP requires a single argument which must be a register name. The instructions in the second and third examples require a single argument which may have any eight bit value. 100 is used as the value of the argument in the second example; the value which PVAL represents is used as the argument value in the third example. In the fourth example the MOV instruction requires two arguments which must be register names. The arguments are separated by a comma. The instructions in the last two examples require two arguments. The first argument must be a register name. The second argument may have any 16 bit value. The value of ADDR is used as the value of the argument in the fifth example; the 16 bit value of the text string XY is used as the argument value in the last example.

COMMENT FIELD

Any line of source code may contain a comment. The comment is optional. It is just a place for you to make a remark about the source code (or anything else, for that matter). The comment field usually contains a running commentary on the operation of the program.

A comment is separated from the instruction or arguments by a tab, a space, or a semicolon. If a line contains nothing but a comment field the comment must start with a semicolon or an asterisk. Here are some examples of source lines with comments.

```
        MOV     A,B     This is a comment.
        CMA             ; This comment starts with semicolon
        MOV     D,A; This comment is separated by semicolon.
; This line contains only a comment.
                ;So does this one.
```

A macro is a named collection of one or more lines of code. After the macro has been defined, it may be inserted into a program one or more times simply by typing the macro's name in place of an instruction. See the ASSEMBLER INSTRUCTION section for more detailed information about macros.

A macro is defined by the instruction MACRO. It must have a name which starts in column one. The body of the macro follows on subsequent lines. The end of the macro definition is indicated by the instruction ENDMAC.

```
|FLIP: MACRO             ; DEFINE A MACRO CALLED FLIP.
|      MOV     A,M       ; GET A BYTE.
|      CMA               ; COMPLEMENT IT.
|      MOV     M,A       ; REPLACE IT.
|      ENDMAC            ; END OF MACRO DEFINITION.
```

This macro may be called in a program by using the name FLIP as an instruction.

```
|      LXI     H,ADDR    ; POINT TO A MEMORY LOCATION.
|      FLIP              ; COMPLEMENT ITS CONTENTS.
```

When the program is assembled, the macro in the preceding example is expanded as follows.

```
|      LXI     H,ADDR    ; POINT TO A MEMORY LOCATION.
|      MOV     A,M       ; GET A BYTE.
|      CMA               ; COMPLEMENT IT.
|      MOV     B,M       ; REPLACE IT.
```

Notice that the comments in the macro definition are stored with the macro text and appear in the listing when the macro is expanded. If your program defines quite a few macros, a lot of storage space may be taken up by comments. You can save this space by starting each comment with two semicolons. This prevents the comment from being stored.

```
|COM:  MACRO
|      MOV     A,M       ; THIS COMMENT IS STORED.
|      MOV     M,B       ;; THIS COMMENT IS NOT.
|      ENDMAC            ; END OF MACRO DEFINITION.
```

This macro is expanded as follows:

```
|      MOV     A,M       ; THIS COMMENT IS STORED.
|      MOV     M,B
```

A macro may be defined with dummy arguments which are replaced with real arguments when the macro is called later in the rogram. The dummy arguments are listed on the first line of the macro as arguments separated by commas. Each time a dummy argument is encountered in the body of the macro, it is replaced with a numbered marker.

When the macro is called, the real arguments are given on the call line as arguments separated by commas. The first real argument replaces every occurrence of the first marker in the macro body, the second replaces the second, etc. If there are too many real arguments the extras are ignored. If there are not enough real arguments the missing ones are treated as null arguments, that is, arguments without any characters in them.

```
|OUTPUT:MACRO  PORT,ADDR; DEFINE MACRO CALLED OUTPUT.
|       LDA     ADDR     ; GET CONTENTS OF MEMORY LOCATION.
|       OUT     PORT     ; TRANSMIT TO OUTPUT PORT.
|       ENDMAC           ; END OF MACRO DEFINITION.
```

The macro is called as follows:

```
|       OUTPUT  27H,DATA; TRANSMIT A BYTE FROM DATA TO OUTPUT PORT 27.
```

It is expanded like this:

```
|       LDA     DATA     ; GET CONTENTS OF MEMORY LOCATION.
|       OUT     27H      ; TRANSMIT TO OUTPUT PORT.
```

The dummy arguments may occur anywhere in the macro body, including the label and instruction fields.

```
|MACK: MACRO   LAB,INS,ARG1,ARG2
|LAB:  INS     ARG1,ARG2
|      ENDMAC
```

This macro is called as follows:

```
|       MACK    ABC1,MOV,A,M
```

It is expanded as follows:

```
|ABC1: MOV     A,M
```

Dummy symbols are treated like ordinary symbols. They must start with a letter, $, ., %, #, or _. Only the first eight characters are significant. However, the arguments which replace the markers when the macro is expanded may contain any number of characters including quoted commas.

A dummy argument may be concatenated with text in the macro body by using the ! as a concatenation character. Whenever ! immediately precedes or follows a dummy symbol in the macro body, the ! and the dummy symbol are both replaced by the marker, without any intervening space. When the macro is later expanded the marker is replaced by a real symbol.

```
|TEXT: MACRO   TAG,TXT
|T!TAG:DB      TXT,0
|      ENDMAC
```

This macro is called as follows:

```
|      TEXT    1,"Hi there, boys and girls"
|      TEXT    2,"This is Uncle Fink"
```

It is expanded as follows:

```
|T1:   DB      "Hi there, boys and girls",0
|T2:   DB      "This is Uncle Fink",0
```

One macro definition may contain another macro definition. The dummy arguments apply to all the macro definitions. The text for the inner (contained) macro definition is modified and stored inside the outer macro body.

```
|OUTER:MACRO   ARG1,ARG2; DEFINE OUTER MACRO.
|      LDA     ARG1
|INNER:MACRO   ARG3     ; DEFINE INNER MACRO.
|      ADI     ARG3
|      ENDMAC           ; END OF INNER MACRO DEFINITION.
|      STA     ARG2
|      ENDMAC           ; END OF OUTER MACRO DEFINITION.
```

At this time OUTER has been defined but INNER has not. A call to INNER results in an error message. INNER is defined when OUTER is called and expanded.

```
|      OUTER   HERE,THERE
```

It is expanded as follows:

```
|      LDA     HERE
|INNER:MACRO   ARG3     ; DEFINE INNER MACRO.
|      ADI     ARG3
|      ENDMAC           ; END OF INNER MACRO DEFINITION.
|      STA     THERE
```

Now INNER has also been defined. It can be called as follows:

```
|POINT:INNER   34
```

Notice that the label POINT has been placed in front of the macro call. It is expanded as follows:

```
|       ADI     34
```

Finally, a macro may contain a call to another macro. In fact, macro expansions may be nested to sixteen levels.

```
|NEST: MACRO   PLACE
|      LDA     PLACE
|      INNER   123      ; NESTED MACRO CALL.
|      STA     PLACE
|      ENDMAC
```

It is called as follows:

```
|       NEST    BOPPER
```

This is expanded as follows:

```
|       LDA     BOPPER
|       ADI     123
|       STA     BOPPER
```

Each argument may be made up of any combination of user defined symbols, numbers, or quoted character strings. They may be combined by + (add), - (subtract or negate), * (multiply), / (divide), and & (logical and). Any argument may be preceded with the word NOT (complement). The arithmetic procedures are carried out from left to right. No parentheses are allowed. For example, 1+2*3 is evaluated as 9, not 7. Arithmetic symbols may not be combined. For example, SYM1&NOT SYM2 causes an error. To prevent the error, divide the operation into two lines. The first line is NSYM2 EQU NOT SYM2. The second line contains SYM1&NSYM2.

RELOCATABLE SYMBOL ARITHMETIC

Absolute symbols may be used in all arithmetic operations. For example, the following operations are all valid:

```
        ABS
OFFSET  EQU     27
        LDA     $+OFFSET
        STA     TABLE-OFFSET
         .
         .
TABLE:  DS      100
```

Relocatable symbols may be used in some arithmetic operations but not in others. A constant (absolute) symbol may be added to or subtracted from a relocatable symbol. The result of the operation belongs to the same section as the relocatable symbol. A relocatable symbol may not be multiplied, divided, anded, or NOTed.

```
        PROG
        LDA     TABLE-3         Valid
        LXI     H,TABLE/4       Not valid
         .
         .
TABLE:  DS      100
```

A relocatable symbol may be subtracted from another relocatable symbol of the same section. The result is the absolute difference between the two symbols. The two symbols may not be in different sections because the addresses represented by the symbols are not known until the module is linked.

```
        DATA
TABLE:  DB      'A'
         .
         .
        DB      'Z'
LENGTH  EQU     TABLE-$ ; NUMBER OF BYTES IN TABLE
```

The assembler evaluates an expression from left to right. In the following example the first two terms are relocatable but the result of the subtraction is an absolute number which may be divided by another absolute number.

```
        DATA
TABLE:  DW      BLOTZ
          .
          .
        DW      BLINTZ
LENGTH  EQU     TABLE-$/2 ; NUMBER OF ADDRESSES IN TABLE
```

An external symbol may not be used in any arithmetic or logical operation.

```
        EXT     BLOTZ
        LDA     BLOTZ    Valid
        STA     BLOTZ+3  Valid
        LHLD    3-BLOTZ  Not valid
```

STRINGS

A quoted character string must start with either a single quote (') or a double quote ("). The quote character is used as a delimiter to determine the end of the string. All characters in the string up to but not including the second delimiter are evaluated. Both delimiters must be the same. If the second one is missing, all remaining characters up to the end of the line are considered part of the quote string. For example, DW 'AB' is evaluated as 4142H.

NUMBERS

Some instructions require a single byte argument. If the value of the evaluated argument requires more than one byte to express, an error message is printed. For example, 260 is evaluated as 104H. MVI A,260 gives an error message. The exception to this rule is a number whose high byte is 0FFH, such as -2 (0FFFEH). This number returns only the low byte without an error message.

Numbers may be represented in binary, octal, decimal, or hex notation. All numbers must start with a decimal digit (0 - 9). That is, a hex number that starts with a letter should have a zero before it, or it will be interpreted as a symbol (0FFH). If the number is not a decimal number it must end with a letter to indicate the notation.

TYPE	DIGITS	TERMINATION
Binary	0 - 1	B
Octal	0 - 7	O or Q
Decimal	0 - 9	D or . or nothing
Hex	0 - 9, A - F	H

Here are some examples of proper numbers:

1011001B	Binary
1357Q	Octal
2246O	
2468.	Decimal
1234D	
99	
3B9CH	Hex
0FFFH	

RELATIVE JUMPS

The relative jump instructions require an argument which is evaluated as a 16 bit address. The difference between the address and two plus the current value of the program counter is used as the eight bit signed relative jump offset. If the offset cannot be expressed by an eight bit number, that is, if the address is farther than plus or minus 127 bytes from the program counter plus two; the jump cannot be made and an error message is printed. A relative jump may start and end in the same relocatable section but it may not jump from one section to another.

REGISTER NAMES

Single (eight bit) registers have the following names:

A	
B	
C	
D	
E	
H	
L	
M	
d(IX)	
d(IY)	
I	Interrupt vector register
R	Memory refresh register

M is a memory location whose address is in the HL register pair, that is, HL points to register M. Memory locations d(IX) and d(IY) are locations whose address is the contents of the IX or IY register added to d where d is a signed eight bit number. The symbol d can be evaluated as a signed eight bit number. It may also be omitted altogether.

Double (16 bit) registers have the following names:

B	BC pair
D	DE pair
H	HL pair
PSW	Processor status word, A and flags
SP	Stack pointer
IX	Index register X
IY	Index register Y

P may be substituted for PSW, S may be substituted for SP, and X or Y may be substituted for IX or IY in any instruction.

This section contains the machine instructions organized into logical groups. They generate code which tells the computer what to do. The first line of the description of each group of instructions is an example of the proper use of an instruction in the group.

Format: JMP BLOTZ

The jump and call instructions require an argument which is evaluated as a 16 bit address.

JMP	Jump.
JNZ	Jump if non-zero.
JZ	Jump if zero.
JNC	Jump if no carry.
JC	Jump if carry.
JNV	Jump if no overflow.
JV	Jump if overflow.
JPO	Jump if parity is odd.
JPE	Jump if parity is even.
JP	Jump if positive.
JM	Jump if minus.

JNV generates the same code as JPO. JV the same as JPE.

JR	Jump relative.
JMPR	Jump relative.
JRNZ	Jump relative if non-zero.
JRZ	Jump relative if zero.
JRNC	Jump relative if no carry.
JRC	Jump relative if carry.
DJNZ	Decrement B and jump relative if B <> 0.

Format: CALL BLOTZ

CALL	Call a subroutine.
CNZ	Call if non-zero.
CZ	Call if zero.
CNC	Call if no carry.
CC	Call if carry.
CNV	Call if no overflow.
CV	Call if overflow.
CPO	Call if parity is odd.
CPE	Call if parity is even.
CP	Call if positive.
CM	Call if minus.

CNV generates the same code as CPO. CV the same as CPE.

RETURN

Format: RET

The return instructions do not require an argument.

RET	Return from a subroutine.
RNZ	Return if non-zero.
RZ	Return if zero.
RNC	Return if no carry.
RC	Return if carry.
RNV	Return if no overflow.
RV	Return if overflow.
RPO	Return if parity is odd.
RPE	Return if parity is even.
RP	Return if positive.
RM	Return if minus.

RNV generates the same code as RPO. RV is the same as RPE.

RETI	Return from interrupt.
RETN	Return from non-maskable interrupt.

RESTART

Format: RST 3

The restart instructions require an argument which represents a number between zero and seven.

RST n Restart at location n*8 where n is a value from 0 - 7.

ACCUMULATOR

Format: ADI 27

The accumulator immediate instructions require an argument which is evaluated as eight bits. These instructions modify all flags. All instructions except CPI leave the result of the operation in the A register. The CPI instruction does not change the A register.

ADI	Add immediate.
ACI	Add immediate with carry.
SUI	Subtract immediate.
SBI	Subtract immediate with borrow.
ANI	AND immediate.
XRI	Exclusive OR immediate.
ORI	OR immediate.
CPI	Compare immediate.

Format: ADD 3(IX)

The accumulator register instructions require an argument which is a single register name, A, B, C, D, E, H, L, M, d(IX), or d(IY). These instructions modify all flags. All instructions except CMP leave the result of the operation in the A register. The CMP instruction does not change the A register.

ADD	Add register to A.
ADC	Add register to A with carry.
SUB	Subtract register from A.
SBB	Subtract register from A with borrow.
ANA	AND register with A.
XRA	Exclusive OR register with A.
ORA	OR register with A.
CMP	Compare register with A.

INCREMENT, DECREMENT

Format: INR A

The single register increment and decrement instructions require an argument which is a single register name, A, B, C, D, E, H, L, M, d(IX), or d(IY). All flags except carry are modified.

INR	Increment the register.
DCR	Decrement the register.

Format: INX H

The double register increment and decrement instructions require an argument which is a double register name, B, D, H, SP, IX, or IY. No flags are modified.

INX	Increment the register pair.
DCX	Decrement the register pair.

DOUBLE ADD, SUBTRACT

Format: DAD B

The double register add and subtract instructions require an argument which is a double register name, B, D, H, or SP. DADX accepts IX instead of H as an argument and DADY accepts IY instead of H as an argument. The DADC and DSBC instructions modify all flags. The other instructions modify only the carry flag.

DAD	Add the register pair to HL.
DADC	Add the register pair to HL with carry.
DSBC	Subtract the register pair from HL with borrow.
DADX	Add the register pair to IX.
DADY	Add the register pair to IY.

LOAD, STORE

Format: LDAX B

The LDAX and STAX instructions require an argument which is a double register name, B, or D.

LDAX	Load A from location pointed to by register pair.
STAX	Store A in location pointed to by register pair.

Format: LDA BLOTZ

The load and store direct instructions require an argument which is evaluated as a 16 bit address.

LDA	Load A.
LBCD	Load BC.
LDED	Load DE.
LHLD	Load HL.
LSPD	Load stack pointer.
LIXD	Load IX.
LIYD	Load IY.
STA	Store A.
SBCD	Store BC.
SDED	Store DE.
SHLD	Store HL.
SSPD	Store stack pointer.
SIXD	Store IX.
SIYD	Store IY.

PUSH,POP

Format: PUSH H

The push and pop instructions require an argument which is a double register name, B, D, H, PSW, IX, or IY.

PUSH	Push the register pair onto the stack.
POP	Pop the stack into the register pair.

Format: IN 5

The input and output instructions require an argument which is evaluated as an eight bit port number. These instructions do not modify any registers.

IN	Move data from the input port into A.
OUT	Move data from A to the output port.

Format: INP D

The input register and output register instructions require an argument which is a single register name, A, B, C, D, E, H, L, or M. The OUTP instruction does not modify any flags. The INP instruction modifies all flags except carry. The INP M instruction only modifies the flags, not the memory location.

INP	Move data from the input port whose port number is in C into the register.
OUTP	Move data from the register to the output port whose port number is in C.

Format: INI

The input memory and the output memory instructions do not require an argument. The zero flag is set if the B register is decremented to zero. The carry flag is not affected.

INI	Move data from the input port whose port number is in C into M. Decrement B. Increment HL.
INIR	Do INI until B = 0.
IND	Same as INI except decrement HL.
INDR	Do IND until B = 0.
OUTI	Move data from M to the output port whose port number is in C. Decrement B. Increment HL.
OUTIR	Do OUTI until B = 0.
OUTD	Same as OUTI except decrement HL.
OUTDR	Do OUTD until B = 0.

MOVE, LOAD IMMEDIATE

Format: MVI B,27

The move immediate instructions require two arguments; a single register name, A, B, C, D, E, H, L, M, d(IX), or d(IY), and an argument which is evaluated as eight bits. The two arguments are separated by a comma.

MVI Move the number into the register.

Format: LXI H,BLOTZ

The load immediate instructions require two arguments: a double register name, B, D, H, SP, IX, or IY, and an argument which is evaluated as 16 bits. The two arguments are separated by a comma.

LXI Load the number into the register pair.

Format: MOV A,B

The move instructions require two arguments. Both are single register names, A, B, C, D, E, H, L, M, d(IX), or d(IY). The arguments are separated by a comma. The two arguments should not both be memory, that is, you can't say MOV M,(IX).

MOV Move second register into first register.

Format: LDI

The block move and compare instructions do not require an argument. The P/V flag is cleared to zero if BC is decremented to zero. The load instructions modify only the P/V flag. The compare instructions set the zero flag if the contents of A equal the contents of M and also modify the sign flag. These instructions do not change the carry flag.

LDI	Move contents of memory pointed to by HL into memory pointed to by DE. Increment DE and HL. Decrement BC.
LDIR	Do LDI until BC = 0.
LDD	Same as LDI except decrement DE and HL.
LDDR	Do LDD until BC = 0.
CCI	Compare A with M. Increment HL. Decrement BC.
CCIR	Do CCI until BC = 0 or A = M.
CCD	Same as CPI except decrement HL.
CPDR	Do CPD until BC = 0 or A = M.

Format: BSET 3,M

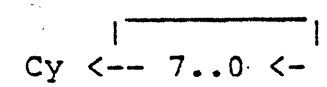
The bit set, reset, and test instructions require two arguments: an argument which represents a bit position between zero and seven, and a single register name, A, B, C, D, E, H, L, M, d(IX), or d(IY). The arguments are separated by a comma (bit number, register name). Only the BIT instruction modifies any registers. The carry flag is not changed.

BSET	Set the bit in the register.
RES	Reset the bit in the register.
BIT	Copy the bit in the register into the zero flag.

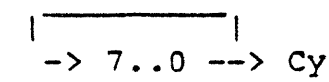
Format: RLC

The rotate A instructions do not require an argument. They modify only the carry flag.

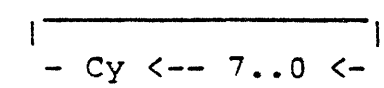
RLC Rotate A left 8 bits. MSB into carry.



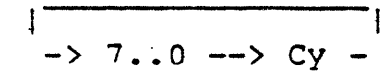
RRC Rotate A right 8 bits. LSB into carry.



RAL Rotate A, carry left 9 bits. MSB into carry.
RLA Same as RAL.



RAR Rotate A, carry right 9 bits. LSB into carry.
RRA Same as RAR.



Format: RLCR D

The rotate and shift instructions require an argument which is a single register name, A, B, C, D, E, H, L, M, d(IX), or d(IY). These instructions modify all flags.

RLCR	Rotate register left 8 bits. See RLC.	MSB into carry.
RLAR	Rotate register left 9 bits. See RAL.	MSB into carry.
RRCR	Rotate register right 8 bits. See RRC.	LSB into carry.
RRAR	Rotate register right 9 bits. See RAR.	LSB into carry.
SLAR	Shift register left 9 bits. Cy <- 7..0 <- 0	0 into LSB. MSB into carry.
SRAR	Shift register right 9 bits. |____| -> 7..0 -> Cy	Sign into MSB. LSB into carry.
SRLR	Shift register right 9 bits. 0 -> 7..0 -> Cy	0 into MSB. LSB into carry.

Format: RLD

The rotate digit instructions do not require an argument. These instructions modify all flags except carry.

RLD Rotate four LSBs of A left with M.

RRD Rotate four LSBs of A right with M.

Format: CMA

Several miscellaneous instructions do not require an argument. No flags are affected unless otherwise noted.

CMA	Complement accumulator.
NEG	Negate accumulator. All flags modified
DAA	Decimal adjust accumulator. All flags modified.
STC	Set carry. Only carry modified.
CMC	Complement carry. Only carry modified.
NOP	No operation.
HLT	Halt.
EXAF	Exchange A 1, flags 1 with A 2, flags 2.
EXX	Exchange BC 1, DE 1, HL 1 with BC 2, DE 2, HL 2.
XTHL	Exchange the contents of the top of the stack with HL.
XTIX	Exchange the contents of the top of the stack with IX.
XTIY	Exchange the contents of the top of the stack with IY.
XCHG	Exchange DE with HL.
PCHL	Load the program counter from HL.
PCIX	Load the program counter from IX.
PCIY	Load the program counter from IY.
SPHL	Load the stack pointer from HL.
SPIX	Load the stack pointer from IX.
SPIY	Load the stack pointer from IY.
DI	Disable interrupts.
EI	Enable interrupts.
LDAI	Load A with I. Zero and sign flags modified. P/V flag gets contents of IFF.
STAI	Store A in I.
LDAR	Load A with R. Zero and sign flags modified. P/V flag gets contents of IFF.
STAR	Store A in R.
IM0	Set interrupt mode 0.
IM1	Set interrupt mode 1.
IM2	Set interrupt mode 2.

ASSEMBLER INSTRUCTIONS

This section contains assembler instructions. They tell the assembler what to do. In some cases they generate machine code. The first line or lines of the description of each instruction is an example of the proper use of the instruction.

Format:
```
BLOTZ: MACRO  REG
       SLAR   REG
       ENDMAC
```

A macro definition requires the MACRO instruction with a label, zero or more lines of code which are stored as the body of the macro definition, and an ENDMAC instruction, which marks the end of the macro body. The line containing the MACRO instruction may also contain several dummy arguments separated by commas. A macro definition may contain other macro definitions (255 maximum) and calls to other macros (15 maximum).

Once a macro has been defined it may be called by using the macro name in place of an instruction. The code stored for that particular macro is recalled and entered in the program, character by character, and evaluated.

When the MACRO instruction is encountered, the label is entered in the user's symbol table and marked as a macro. The dummy argument symbols are stored in a temporary symbol table. The code in the body of the macro definition is stored character by character in the macro storage space. Comments beginning with two semicolons are not stored. If a symbol in the body is encountered which matches one of the dummy argument symbols, a numbered marker is stored in the macro storage space instead of the symbol. If the symbol matches the first dummy symbol the marker is given the value one, if it matches the second symbol it is given the value two, etc. The exclamation point (!) is used as a concatenation character. If a dummy symbol in the body is preceded or followed by the concatenation character, the ! is removed along with the dummy symbol when it is replaced by a marker. The macro definition may contain one or more embedded macro definitions. The dummy argument symbols are compared to symbols in all levels of the definition. All dummy symbols are replaced by markers.

The line containing the macro call may also contain one or more arguments separated by commas. These arguments (actually character strings) are substituted for the markers in the macro body. The arguments may be any length (as long as they all fit on one line), and may contain commas in quoted strings. The first argument string replaces every occurrence of the first marker, the second string replaces the second marker, etc.

DEFINE BYTE, WORD

Format: DB 'ABC'

The DB (Define Byte) and DW (Define Word) instructions may be followed by one or more arguments. Each argument is evaluated as a separate byte or word. If a DB argument is a text string enclosed in single or double quotes, the seven bit ASCII value of each character in the string is returned.

```
EXPRESION          CODE GENERATED

DB 100             64
DB 'MOM'           4D
                   4F
                   4D
DW 100             64
                   00
DW 1234H,4567H     34
                   12
                   67
                   45
```

Format: DBS 'AB',CR,LF

The DBS (Define Byte Sign) and DBZ (Define Byte Zero) instructions are similar to the DB instruction. They differ in the way they treat the termination of the command line. The DBS instruction sets the sign bit of the last character in the line. The following pairs of lines generate the same code:

```
DB      'ABCDE','F'+128
DBS     'ABCDEF'

DB      'Hi there',CR,LF+128
DBS     'Hi there',CR,LF
```

The DBZ instruction appends a zero byte to the end of the line. The following pairs of lines generate the same code:

```
DB      'ABCDEF',0
DBZ     'ABCDEF'

DB      'Hi there',CR,LF,0
DBS     'Hi there',CR,LF
```

DEFINE STORAGE

Format: DS 200

The DS (Define Storage) instruction requires one argument and reserves the amount of space (in bytes) determined by the value of the argument. The instruction does not generate any code. The instruction is used to allocate space in memory for variables and tables without specifying the contents of those locations or generating any code in the HEX or BIN files. For example, assume SIZE represents the value 100.

```
|       DS      SIZE    Reserve 100 bytes of space in memory.
|       DS      14      Reserve 14 more bytes.
```

Format:
```
IF   KFLAG
CALL BLOTZ
ENDIF
```

The IF instruction requires one argument. If the value of the argument is zero, assembly of code is suppressed until an ELSE or ENDIF instruction is encountered at which time it resumes. If the value is non-zero, assembly continues until an ELSE instruction is encountered. Then, assembly is suppressed until an ENDIF instruction is encountered. The use of the ELSE instruction is optional. For example, assume SWITCH is equal to zero.

```
|       IF      SWITCH      Argument evaluates to zero.
|       INR     A           Don't assemble this code.
|       ELSE
|       DCR     A           Assemble this code instead.
|       ENDIF
|
|       IF      NOT SWITCH  Argument evaluates to FFFF.
|       DCR     A           Assemble this code.
|       ENDIF
|
|       MOV     C,A         Always assemble this code.
```

IF instructions (with optional ELSEs) may be nested to 255 levels.

ENTRY, EXT

Format: ENTRY SIN,COS

The ENTRY instruction requires one or more arguments which are symbol names. It marks those symbols as entry points. The symbols must be defined somewhere in the program (used as a label, for instance). Entry point symbols are passed via the relocatable output file (REL file) to the linker to define the symbols for use by other modules. This instruction may be used anywhere in the program. The entry instruction is not valid when the assembler is generating a hex or binary file.

Format: EXT TAN,COT

The EXT instruction requires one or more arguments which are symbol names. It tells the assembler that those symbols are not defined in the current program but will be defined later in other modules. EXT symbols are passed via the REL file to the linker to be defined by entry point symbols in other modules. This instruction may be used anywhere in the program. The EXT instruction is not valid when the assembler is generating a hex or binary file.

ABS, PROG, DATA, COM

Format: ABS

The ABS, PROG (REL may be used instead of PROG), and DATA instructions do not require an argument. They tell the assembler to begin or continue generating code in a particular section. If code had been generated in that section before, the program counter points to the next available byte of storage so that code generation continues from where it left off last time. These instructions are not valid when the assembler is generating a hex or binary file.

Format: COM BLOTZ

The COM instruction may take an eight character name as an argument. If no name is given it is assumed to be blank (all spaces). It tells the assembler to begin or continue generating code in that common section in exactly the same way as the ABS, PROG, and DATA instructions do. There may be as many as fifteen different common sections. The COM instruction is not valid when the assembler is generating a hex or binary file.

ORG, LOAD

Format: ORG 100H

The ORG instruction requires an argument which is evaluated as a 16 bit address. The instruction sets the assembler, HEX, and BIN program counters to that address; that is, it determines the starting address of the next block of code generated. The type of the argument (section in which it was defined) determines the type of the new section. For example, if GRIBLY was defined in the data section:

```
        ORG     GRIBLY+100
```

tells the assembler to continue generating code in the data section.

```
        ORG     20
```

has an absolute argument and tells the assembler to generate code in the absolute section.

If the line containing the ORG instruction contains a label, the label is set to the new value of the program counter.

```
GUM:  ORG     123       GUM has the value 123.
```

If you are generating a COM file you may not ORG below 100H + BOOT and you may not ORG backwards (ORG to a location less than the current program counter).

Format: LOAD 1000H

The LOAD instruction is only valid when the assembler is generating hex code. It is not valid when the assembler is generating relocatable code or COM file code. It requires an argument which is evaluated as a 16 bit address. The instruction forces the code generated by the assembler to be loaded into memory whose address is different from the address set by the ORG instruction. This allows you to load code into one region of memory and later move it to another region for execution (for example, programming a PROM). The LOAD instruction requires an argument. It sets the BIN and HEX program counter to the value of the argument but does not change the assembly program counter. For example, if you were writing code to be loaded at 24H but executed at 1003H you would use the instructions:

```
|       ORG     1003H   Set assembler program counter to 1003H.
|       LOAD    24H     Set binary and hex program counter to 24H.
|LOOP:  DCR     C       0D is stored at 24H.
|       JNZ     LOOP    C2 is stored at 25H.
|                       03 is stored at 26H.
|                       10 is stored at 27H.
```

NAME

Format: NAME TRIG

The NAME instruction requires an eight character name as an argument. This name is passed via the relocatable file to the linker and appears in the module name listing. This instruction may be given more than once in a program but only the name specified last is put in the REL file. If this instruction is not used in a program the first eight characters of the REL file name are used as the module name. The NAME instruction is not valid when the assembler is generating a hex or binary file.

INCLUDE

Format: INCLUDE <filename>

INCLUDE temporarily changes the input file to the assembler. This allows code in another file to be inserted into a program during assembly. When the INCLUDED file is exhausted, the assembler resumes reading the source lines from the original source file with the line immediately after the INCLUDE instruction.

Note that nested INCLUDE files are not permitted (I.E. a file which is an argument to the INCLUDE instruction may not contain any INCLUDE instruction).

LIBFILE

Format: LIBFILE ALTLIB

The LIBFILE instruction requires an eight character name as an argument. This name is passed via the relocatable file to the linker and tells the linker to use the file given by this command (with an assumed extension REL) as the library file. If no LIBFILE command is given the linker uses the default library file, LIB.REL. This instruction may be given more than once in a program but only the LIBFILE name specified last is put in the REL file. The LIBFILE instruction is not valid when the assembler is generating a hex or binary file.

Format: CHAR EQU 'Z'

The EQU instruction requires a label and an argument which is evaluated as a 16 bit number. The label is given the 16 bit value. A symbol (the label) may be defined only once in a program with the EQU instruction.

Format: CHAR SET 'X'

The SET instruction is similar to the EQU instruction. It requires a label and an argument which is evaluated as a 16 bit number. The label is given the 16 bit value. The SET instruction may be used to change the value of a symbol (the label) as often as desired.

END

Format: END BLOTZ

The END instruction may be placed at the end of a program but its use is optional. The END statement may have one argument (optional) which is evaluated as a 16 bit address. The value of the argument is used by the operating system as the starting address of the program. The starting address must be in an ABS, PROG, or DATA section. If it is in an EXT or COM section an error message is printed and the starting address is ignored. If no starting address is given, the operating system is able to load the program but not start it. If a starting address is given with the ORG address not equal to the LOAD address, an error message is printed and the starting address is ignored. (A program cannot be executed properly unless it is loaded at its execution address.)

```
|       END             Program has no starting address.
|       END     22H     Program is started at 22H.
|       END     GUMBAL  Program is started at GUMBAL.
```

LIST, NLIST, MTLIST, NMTLIST

Format: NLIST

The NLIST and LIST pseudo-ops turn the listing off and back on. When NLIST is encountered it suppresses the listing. When LIST is encountered it reenables the listing.

```
|       NLIST
|       MOV     A,B     Assemble this code but don't list.
|       MOV     D,E
|       LIST
|       POP     H       Resume listing.
```

The NMLIST and MLIST pseudo-ops turn the listing of macro definitions and expansions off and back on. When NMLIST is encountered it suppresses the listing of lines containing either macro definitions or macro expansions. When MLIST is encountered it reenables the listing.

Format: MTLIST

The NMTLIST and MTLIST pseudo-ops turn the listing of the text part of macro expansions off and back on. When NMTLIST is encountered it suppresses the listing of the text part of macro expansions (the bodies of the macros), but does not suppress the listing of the hex code generated by the macros. When MTLIST is encountered it reenables the listing.

Argument too big	The value of the argument is greater than 255 or less than -255.
	The value of an argument in an RST instruction is greater than seven.
Bad argument	An unknown character, number, or symbol is used in an argument.
	IX or IY may not be used as an argument with this instruction.
Bad arithmetic operator	An unknown character is used as an arithmetic operator.
Bad base	The starting address is in a section other than ABS, PROG, or DATA.
Bad instruction	An entry in the instruction field is not recognized as an instruction or macro.
Bad label	The label does not start with a $, %, ., or letter.
Bad number	The radix character is unknown.
	An improper digit appears in the number.
Bad symbol	The symbol does not start with a $, %, ., or letter.
Can't back up in COM file	Attempted to ORG to a value less than the current value of the program counter or less then 100H. Code in a COM file can only go forward.
Displacement too big	The value of the displacement is greater than 127 or less than -128.
Division by 0	Attempted division by zero.
Dummy redefined	A dummy argument in the macro definition is used more than once.

Extra argument	Too many arguments are given for this instruction.
Extra ELSE	The ELSE instruction does not have a matching IF instruction.
Extra ENDMAC	The ENDMAC instruction does not have a matching MACRO instruction.
File not found	The INCLUDE file cannot be found.
Macro not defined	A macro is called before it is defined.
MACRO symbol	A macro name is used in an instruction argument.
Missing argument	Not enough arguments are given for the instruction.
Missing)	The) is missing from the name of an index register.
Multiple tag	This label has been used before.
Nested INCLUDE	The INCLUDE file calls another INCLUDE file.
No EQU label	The EQU instruction does not have a label.
No expression	An expression is not allowed with this instruction, only a symbol.
No EXT	An external symbol may not be used with this instruction.
No MACRO label	The macro definition does not have a label.
No relocate	A relocatable symbol may not be used with this instruction or arithmetic operation. If the assembler is generating an absolute binary or hex file a relocatable operation is not allowed. A relative jump instruction jumps from one relocatable section to another.

No SET label	The SET instruction does not have a label.
Not allowed in COM file	The LOAD instruction cannot be used when generating COM file. Generate a HEX file instead.
Offset not zero	The starting address is given with the LOAD address not equal to the ORG address.
Out of range	The destination is too far for a relative jump.
Redefined	The value of the label is changed.
	A macro name is used as a non-macro label.
String too long	The string contains more than two characters.
Symbol not found	An undefined symbol is used in an argument.
Symbol table full	There is no more room to add symbols to the symbol table or to define more macros.
Too many arithmetic operators	More than one arithmetic operator is used in front of a symbol or number.
Too many commons	More than 15 common sections have been defined.
Too many externals	More than one external symbol has been used in an expression.
Too many index registers	An index register is specified for both arguments in a MOV instruction.
Too many macro nest levels	More than 15 macro definitions or 255 macro expansions are nested.

This section contains assembler listings of three modules. The first module contains the main part of the program which reads a string of characters from the keyboard and prints them. The second and third modules contain subroutines which communicate with either the CP/M operating system (second module) or the K3 operating system (third module). This program may be run with either operating system simply by linking the main module with the appropriate subroutine module.

```
                ; String Echo.

     000D       CR      EQU     13      ; Carriage return.
     000A       LF      EQU     10      ; Line feed.

     0001       PRINT:  MACRO   TEXT    ; Print a text string.
    -                       LXI     H,TEXT
    -                       CALL    TXTYP
                        ENDMAC

                        EXT     CI,TXTYP,MONITOR

                START:  PRINT   TITLE
0000+21 0026'                       LXI     H,TITLE
0003+CD 0000#                       CALL    TXTYP
0006'21 0000"           LXI     H,BUFFER; Point to the line buffer.
0009'CD 0000#   LOOP:   CALL    CI      ; Get a character.
000C'77                 MOV     M,A     ; Store it.
000D'23                 INX     H       ; Bump pointer.
000E'FE 0D              CPI     CR      ; End of line?
0010'20 F7              JRNZ    LOOP    ; Not yet.  Keep going.
0012'36 0A              MVI     M,LF    ; Add a line feed.
0014'23                 INX     H
0015'36 00              MVI     M,0     ; Mark the end of the line.
                        PRINT   CRLF
0017+21 0036'                       LXI     H,CRLF
001A+CD 0004#                       CALL    TXTYP
                        PRINT   BUFFER  ; Echo the buffer.
001D+21 0000"                       LXI     H,BUFFER
0020+CD 001B#                       CALL    TXTYP
0023'C3 0000#           JMP     MONITOR ; And return to the monitor.

0026'44 65 6D 6FTITLE:  DBZ     'Demo Program',CR,LF,'*'
     20 50 72 6F
     67 72 61 6D
     0D 0A 2A 00
0036'0D 0A 00   CRLF:   DBZ     CR,LF

     0000"              DATA
0000"0080       BUFFER:DS       128     ; String buffer.
     0000'              END     START
```

```
                ; CP/M Operating System Subroutines.
                ; These subroutines talk to the CP/M operating system.

     0001       IOP:    MACRO   FUNCTION; Call an I/O processor function.
    -                       MVI     C,FUNCTION
    -                       CALL    5
                        ENDMAC

     0000       MON     EQU     0       ; Return to the monitor.
     0001       CREAD   EQU     1       ; Read a character.
     0002       CWRITE  EQU     2       ; Write a character.

                        ENTRY   CI,TXTYP,MONITOR

                ; Read a character from the keyboard with echo.
0000'E5         CI:     PUSH    H       ; Save HL.
                        IOP     CREAD
0001+0E 01                      MVI     C,CREAD
0003+CD 0005                    CALL    5
0006'E1                 POP     H
0007'C9                 RET

                ; Write a text string pointed to by HL.
                ; The string ends with a null.
0008'7E         TXTYP:  MOV     A,M     ; Get a character.
0009'23                 INX     H
  0A'B7                 ORA     A       ; Null?
  0B'C8                 RZ              ; Yes.  Quit.
000C'5F                 MOV     E,A     ; Not yet.
000D'E5                 PUSH    H       ; Save pointer.
                        IOP     CWRITE  ; Write character.
000E+0E 02                      MVI     C,CWRITE
0010+CD 0005                    CALL    5
0013'E1                 POP     H
0014'18 F2              JR      TXTYP   ; Keep going.

                ; Return to the monitor.
                MONITOR:IOP     MON
0016+0E 00                      MVI     C,MON
0018+CD 0005                    CALL    5
```

```
                ; K3 Operating System Subroutines.
                ; These subroutines talk to the K3 operating system.

     0001       IOP:    MACRO   FUNCTION; Call an I/O processor function
    -                       CALL    FUNCTION
                        ENDMAC

     D000       MON     EQU     0D000H  ; Return to the monitor.
     D037       CREAD   EQU     MON+37H ; Read a character.
     D03D       CWRITE  EQU     MON+3DH ; Write a character.

                        ENTRY   CI,TXTYP,MONITOR

                ; Read a character from the keyboard with echo.
                CI:     IOP     CREAD
0000+CD D037                        CALL    CREAD
0003'4F                 MOV     C,A
                        IOP     CWRITE  ; Echo.
0004+CD D03D                        CALL    CWRITE
0007'79                 MOV     A,C
0008'C9                 RET

                ; Write a text string pointed to by HL.
                ; The string ends with a null.
0009'7E         TXTYP:  MOV     A,M     ; Get a character.
000A'23                 INX     H
000B'B7                 ORA     A       ; Null?
000C'C8                 RZ              ; Yes.  Quit.
000D'4F                 MOV     C,A     ; Not yet.
000E'E5                 PUSH    H       ; Save pointer.
                        IOP     CWRITE  ; Write character.
000F+CD D03D                        CALL    CWRITE
0012'E1                 POP     H
0013'18 F4              JR      TXTYP   ; Keep going.

                ; Return to the monitor.
                MONITOR:IOP     MON
0015+CD D000                        CALL    MON
                        END
```

RUNNING THE ASSEMBLER UNDER CP/M

To run the assembler type:

ASMBL <fn>.<opts>,<fn>.<opts>,<fn>.<opts> ... /<type>

where

<fn> is a text file with the extension SRC

<opts> is an optional list of options up to three letters long.

first letter:	drive to get source from.
second letter:	drive to send output file to.
third letter:	drive to send listings to. If this letter is omitted, no listing is generated. If the letter is X, the listing is sent to the console instead of the disk.

<type> specifies the type of the output file. It must be /COM, /HEX, or /REL. If no type is specified /COM is assumed.

If more than one file is specified, the files will be assembled as though they were one large file. The order in which they are listed in the command line is the order in which they would appear in this large file (note: no "large file" is actually created). The name of the last input file is used as the name of the output file. If an option is not specified, or if a space is used in place of a letter, the default drive is used. The exception to this is the listing file: If a space is used, a listing file is created on the default drive, if nothing is specified, no file is created. For example:

A>ASMBL INIT,NAVAGAT/HEX

Assemble INIT.SRC with NAVAGAT.SRC. Get both files from drive A and send NAVAGAT.HEX to drive A. No listing file is generated because no listing drive letter was specified.

C>ASMBL INIT.A,NAVAGAT. BX

Assemble the file INIT.SRC on drive A with NAVAGAT.SRC on drive C. Send NAVAGAT.COM to drive B. Send the listing to the console.

The assembler recognizes two additional instructions under the K3 operating system. They are as follows:

Format: JSW 1000H

The JSW instruction only generates code when the assembler is producing a BIN file under the K3 operating system. It requires one argument which is evaluated as a 16 bit number. The value of the argument is used by the operating system as the job status word. If the 1000H bit is set, the program may be started at the starting address with the operating system RUN or START commands. If the 2000H bit is set, the program may be restarted at a location three less than the starting address with the operating system restart command. If the JSW instruction is not given, the operating system assumes a default value for the job status word.

```
|       JSW       1000H  Allow the program to be started
                         but not restarted.
```

Format: VER '1','2','c'

The VER instruction requires three arguments which are evaluated as three ASCII characters. These three characters are stored only in the K3 BIN or K3 HEX file, and are read only by the K3 LIMITS program. It is recommended that the first two characters be used for a two digit version number and that the third character be used for a single revision letter. If your program has only a single digit version number, the first character should be a space.

```
|       VER     ' ','7','b'     ; version 7b.
|       VER     '2','7','x'     ; version 27x.
```

When the assembler is started it asks you for a file specification. The specification is in the following format:

```
DEV:NAME1.BIN(,REL, or HEX),DEV:NAME2.LST=DEV:NAME3.SRC/B/RE/H/L/G/RU/E
```

Not everything in the specification line needs to be typed in. For example, the extensions (BIN, REL, HEX, LST, SRC) are always filled in by the assembler and should not be typed in. This means that the source file must always have a SRC extension. The listing file always has a LST extension, etc.

The first entry in the specification determines the device and file name (if necessary) to which the BIN, REL, or HEX file is sent. If the output device is non-file structured (paper tape punch, for example), a file name is not needed. If the output is sent to a file structured device and the file name is not given, it is given the name of the last source file.

The /B, /RE, or /H option determines which file is generated, BIN, REL, or HEX. If no option is specified /B is assumed. If no device and file name is specified but the /B, /RE, or /H option is given a BIN, REL, or HEX file is assumed using the last source file name. Here are some examples of proper file specifications:

```
PP:=BLOTZ       Output is sent to the paper tape punch.
DK3:TRIG=BLOTZ  Output is sent to TRIG.BIN on DK3.
DK0:=BLOTZ      Output is sent to BLOTZ.BIN on DK0.
```

If the /G (get) or /RU (run) options are specified the assembler automatically sets the /B option (clears the /RE and /H options) and generates a BIN file. At the end of the assembly the operating system is asked to get (/G) or run (/RU) the BIN file. If any errors are detected in the assembly, the get or run request is suppressed.

The second entry in the specification determines the device and file name (if necessary) to which the listing file is sent. If the output device is non-file structured (line printer, for example) a file name is not needed. If the output is sent to a file structured device and the file name is not given, it is given the name of the last source file. If the /L option is given without a listing file specification a LST file is assumed with the name of the last source file. The listing entry is always the second entry in the specification line and is separated from the first entry by a comma. If no BIN, REL, or HEX file is desired, the line must start with a comma.

```
|.R ASMBLE DK2:=TEST/RU
```

This command loads and runs the assembler, assembles DK0:TEST.SRC into DK2:TEST.BIN, loads, and runs DK2:TEST.BIN.

The /E option sends error messages to the line printer. This is useful for generating a printed record of assembly errors.

If control C is typed while the program is running, the assembly stops, all files are closed, and control returns to the monitor.

If control O is typed while the program is running, the listing of error messages is suppressed. If any other key is typed, the printing resumes.