```
****************
TABLE OF CONTENTS
****************
```

```
*********************************************
PART I - CROMEMCO RELOCATABLE ASSEMBLER MANUAL
*********************************************
```

CROMEMCO MACRO ASSEMBLER INSTRUCTION MANUAL

```
*************************************************************
CHAPTER 1:  GETTING STARTED IN ASSEMBLY LANGUAGE PROGRAMMING
*************************************************************
```

The purpose of an Assembler is to provide a means of translating easily
understood mnemonics, which represent the instructions of a computer,
into object code which may be loaded into memory and run as a program.
The CROMEMCO Disk-Resident Z-80 Relocatable Macro Assembler is a two-
pass assembler which reads source code from a disk file, assembles it,
and produces a relocatable object and/or a print-listing file. These
files may be sent to any of the disks, suppressed altogether, or sent
to the console (listing file only). The CROMEMCO Relocating Linker/
Loader may then be used to locate the assembled code anywhere in memory.
The completely assembled and linked machine code may be saved in a disk
.COM file for execution as a command program.

The use of a relocatable assembler and linker provides one of the most
versatile ways of creating machine language programs for the computer.
The time saved through their use is well-worth the time spent in
gaining familiarity. These two command files allow one to create and
assemble a number of different modules separately, and then link them
together at run time. Or one can link an assembled user-program to an
already existing library of useful object code files. In addition, one
may assemble programs using a compiler (for examples assemble FORTRAN
programs into machine code using CROMEMCO's FORTRAN Compiler), and link
these object modules to existing machine code modules, programs, or
subroutines. At the same time the final program may be located to run
anywhere in memory.

The CROMEMCO Relocatable Assembler (hereafter called ASMB or the
Assembler) is both a Macro and a Conditional Assembler as well.
A separate chapter of this manual is devoted to these features.
The Macro capability allows the user to very easily generate such
things as multiple blocks of code, design added capabilities for
the Assembler for a particular purpose, and write much shortened
versions of source code by having a Macro library searched for
often-used routines. The Conditional (IF statement) assembly
feature allows blocks of code to either be included or not, depen-
ding on the satisfaction of user-defined conditions. There are
also capabilities for INCLUDing other source code files at assembly
time and declaring other program modules EXTernal to the main pro-
gram, which are then linked to it at run-time. All these features
are described further in the chapters on pseudo-ops.

The CROMEMCO Relocatable Macro Assembler is supplied to the user
on diskette (large or small) under the directory entry "ASMB.COM".
The way the Assembler is called is described in detail in Chapter 2.
A source code file to be assembled must have the three-letter exten-
sion .Z80 to be found by ASMB. To assure correct operation the

Assembler should be used with the following minimum hardware con-

figuration: 32K of contiguous RAM memory beginning at location 0 and
the CROMEMCO 4MHz Z-80 CPU card, along with the CROMEMCO Disk Oper-
ating System (CDOS) hardware and software. When called, ASMB loads
into memory at 100H and begins execution there.

Since most users will be eager to try out some of the features of
ASMB right away, this chapter may be used as a step-by-step beginner's
manual for the composition, assembly, link, and execution of a simple
Z-80 machine language program. The name of the program is "TIMER"
and its purpose is to ring the console bell at approximately half-
second (using 4MHz. clock) intervals as determined by a timer loop.
It will not be necessary for those users familiar with assembly
language programming to read this chapter. These persons may skip
ahead to Chapter 2 at this point.

The first step is to turn on the power to the computer and boot-up
the CROMEMCO Assembler Disk (Model FDA) in drive-A. You will notice
upon typing out the directory that supplied along with the Assembler
(ASMB.COM) is the CROMEMCO Text Editor (EDIT.COM) and CROMEMCO Debug
(DEBUG.COM) programs. We will use the Text Editor to enter our source
code program. The Editor manual is also supplied with the Assembler
package and should be used for questions and reference concerning
EDIT. However, some of the simple commands are explained here
for the benefit of the user who is unfamiliar with the Editor.

Hence, the user can now call EDIT giving the name and three-letter
extension of the file we wish to create by typing the following.
(Note that before typing the command line you should have the CDOS
prompt for the drive you are using, for example "A." for drive-A.)

        EDIT TIMER.Z80

The Editor will then respond with:

        CDOS EDITOR VERS. 00.07
        NEW FILE

The prompt for the Editor is an asterisk, "*", and commands may be
entered any time this prompt is displayed. We now wish to enter
the text of the source program so we use the Insert command of EDIT.
This is done simply by typing the letter "I" followed by a carriage
return (CR). We can then start typing lines of text, ending each
line with a carriage return. Mistakes can be corrected by back-
spacing or can be corrected after we have finished with Insert mode
as explained below. There are four fields which may be used in a
line of source code: labels, opcodes, operands, and remarks. Labels
are followed by a colon and remarks are preceded by a semi-colon.
If there are more than one operand, they are separated by a comma.
The instruction mnemonics or opcodes for the various Z-80 instructions
can be found in the Z-80 CPU Technical Manual published by Mostek
and Zilog along with an explanation of each. Note that in the
following text a tab was used to separate the various fields; this

is done in the Editor by typing the CTRL-I on the console. Also
note that either upper- or lower-case are allowed. We now type
in the source code:

```
        ; This program rings the console bell at approximately
        ;   half-second intervals determined by a timer loop.
        ;
BELL:   EQU     7               ; console bell is ASCII 07
WRITE:  EQU     2               ; write character to console
CDOS:   EQU     5               ; use system call to write
TIMIT:  EQU     2FFH            ; 2 is no. of half-seconds;
        ;                             FF (256) is no. of loops
DURAT:  EQU     0FFH            ; FF (256) is loop duration
        ;
        ; Main Program
        ;
START:  LD      SP,STACK        ; initialize stack pointer
LOOP:   LD      BC,TIMIT        ; B is no. of half-sec.;
        ;                             C is no. of loops
TIM2:   LD      A,DURAT         ; get duration (256)
TIM1:   DEC     A               ; decrement and
        JR      NZ,TIM1         ; loop til zero
        DEC     C               ; decrement loop counter
        JR      NZ,TIM2         ; until zero
        DJNZ    TIM2            ; countdown half-seconds
        LD      E,BELL          ; set-up to ring bell
        LD      C,WRITE         ; set-up to write console
        CALL    CDOS            ; call system
        JP      LOOP            ; loop and repeat
        ;
        ; Stack Area
        ;
BOTTOM: DS      40H             ; allow 64 bytes for stack
STACK:  EQU                     ; current location counter
                                  equals top of stack

        END     START
```

This code should be typed in exactly as it appears here (although
comments may be omitted if desired). When the entire body of text
has been entered, end the Insert mode by pressing ESCape or CTRL-Z.
(You have left Insert mode when you again get the asterisk prompt.)
You now would like to review what you have written to check for
errors. Move the character-pointer to the top of the file by
typing "B <CR>". Then type out your file using the T or P command.
For example the command 10T will type out 10 lines, or 0P will
type out the current page of 23 lines. The S or Substitute command
can now be used to make corrections. The format of the command is

        S<oldtext>^[<newtext>^[<CR>

where the "^[" character is an ESCape. The text string for which
you are substituting must be exclusive; for example the command

```
     SP^[R^[
```

is not good because the first "P" encountered will be changed to an

"R". The following command is much better because the substitution
is for a one of a kind string:

```
     SJP^INZ,TIM1^[JR^INZ,TIM1^[
```

The "^I" is the way EDIT prints CTRL-I's. When all corrections have
been made, you may Exit from the Editor by typing "E <CR>". When
the "A." prompt is again displayed on the console, the created file
will have been saved on the disk under the filename "TIMER.Z80".
If you desire more information about editing files at this point,
refer to the Text Editor Manual for complete descriptions of the
Editor commands. We will now proceed with assembling the file.

The Assembler is called in a similar manner to the Editor. The
command line you should type is

```
     ASMB TIMER
```

The Assembler understands when it receives this command line that
it will find the source file on the current drive, and that it will
place the .REL (relocatable) object file and .PRN (print) listing
file on the current drive as well. Our file "TIMER.Z80" will now
be assembled. When finished, control will again be returned to
CDOS and the "A." prompt given. Just prior to exiting, ASMB will
print on the console:

```
     Errors            0

     Program Length  005A (90)

     end of assembly
```

provided you have made no typing errors in editing the file. If
there are some errors, re-edit the file and correct them as described
above. Then re-assemble as before. The numbers above give the
program length first in hexadecimal and then decimal.

The Assembler will now have created the .REL and .PRN files on the
disk. If you would like a listing of the programs type:

```
     TYPE TIMER.PRN
```

and press CTRL-P (assuming you have a printer correctly hooked up
to parallel port 54H; if you have no printers omit the CTRL-P)
before typing the carriage return. The listing will then be printed
on both the console and the printer. There is a great deal of
information contained in this listing. Briefly, the listing consists
of these sections. The first column is the hexadecimal address of
the instruction, and the second column may be one of three things:

(1) the object code of up to a four-byte instruction in hex, (2) the object code of four bytes of data in hex, or (3) the equivalent value of the operand expression in parentheses. The third column gives the line numbers of the source in decimal. The fourth, fifth, and sixth columns are the label, opcode, and operand fields, respectively.

The rest of each line contains the remark if there is one. The complete listing which results from the assembly of our given example source file is given in Chapter 7 along with a detailed description of every feature of the listing.

The last step prior to running the program is to load it into memory. This is done using the CROMEMCO Linker/Loader. The command line that should be typed is:

        LINK TIMER

The Linker will then prompt with an asterisk (*). This means that it is awaiting further instructions. At this point you may either start execution or exit to CDOS, save the file, and execute it as a command file. Let us choose the second method. (For those who wish to try the first method, simply type /G to the "*" prompt.) To the asterisk type the characters /E which will exit to CDOS. LINK will then print on the console a message similar to:

        [1000   105A   16]

The first number is the starting address for execution; the second number is one more than the highest address used by our program, and the last number gives the number of pages to be saved to create a command file. We now create this .COM file by typing:

        SAVE TIMER.COM 16

Command files may be executed directly from CDOS simply by typing their name. They are then loaded into memory beginning at 100H and execution begins there. The Linker has already placed the necessary "JP 1000H" for us at 100H so we execute the TIMER program simply by typing the word "TIMER". The bell on your console should. begin ringing at approximately half-second intervals, telling you that the machine language program we have created is working!

Good assembly language programming practice usually dictates that a program should be debugged before executing it directly as we have done. By this method the user can insert breakpoints to stop execution so that the registers and memory contents can be checked to determine if the program is executing correctly. We have skipped the debugging stage so as not to complicate the example unnecessarily. However, when you create assembly language programs of your own, you can use the CROMEMCO Debugger program (DEBUG.COM) to execute it using breakpoints. To do this you would first have to save your program in a file as we have done above; then load it using DEBUG. See Part III on the Debugger for several examples of the way to debug a program.

The example.program of this chapter is KNOWN TO WORK if the source
is created and assembled according to the procedures outlined
above. Should you have any difficulties with any of the steps,
try working through that step a second time. You may also refer
to the manual which describes that function for a detailed description

of the procedure. This book is divided into a number of distinct
parts, which are listed in the table of contents for reference.
Parts I, II, and III are the complete Assembler, Link, and Debug
manuals, respectively. Part IV is the CDOS Programmer's Manual,
describing the many system calls which can be made to CDOS for
I/O and disk operations. Part V is a description of the Library
of relocatable modules which are supplied with the Assembler disk.
Finally, Part VI is a description of the method for changing the
Logical Unit Number Table (LUN Table) for CROMEMCO FORTRAN IV to
accommodate different I/O drivers. This section is included with
this book because the procedure requires that you use the CROMEMCO
Relocatable Assembler and Linker.

```
*********************************
CHAPTER 2:  CALLING THE ASSEMBLER
*********************************
```

The Assembler is called from disk simply by typing "ASMB" followed
by the filename of the source code to be assembled. This source-
file MUST have the extension .Z80 to be found by the Assembler,
regardless of whether or not it consists entirely of Z80 code.
However, when calling ASMB, the user may specify an optional 3-letter
drive-request for the filename which has NO relation to the 3-letter
extension of the filename on disk. Note that if this 3-letter drive-
instruction is omitted, ASMB will default to the CURRENT drive for all
operations. This drive-request instruction is of the form .GRS, where
G stands for one of the letters, A, B, C, or D, and is the drive on
which the SOURCE file is to be found; R stands for one of the letters,
A, B, C, D, or Z, and is the drive on which the relocatable OBJECT
file is to be placed during assembly (Z means do not create an object
file); and S stands for one of the letters, A, B, C, D, X, or Z, and
is the drive on which the print-listing will be placed during assembly.
In the case of the print-listing Z means do not create the listing, and
X means send the listing to the console but not to the disk. Note that
you may use the Control-P (^P) function of CDOS, as always, to cause
the console listings to also be sent to a printer. Also note that the
relocatable object file will be placed on the disk with the extension,
.REL, and the print-listing will be placed with the .PRN extension.

An example will serve to illustrate further the features described
above. Suppose the file to be assembled resides on disk drive A under
the filename USERFILE.Z80. If it is desired not to have the .REL and
.PRN files sent to drive A (for lack of room an disk A, for example),

the Assembler might be called by the command line:

          ASMB USERFILE.ABX <^P> <CR>

This will assemble the source file on drive A, create an object file on
drive B, and send the print-listing to both the console and the printer.

A number of options may also be specified at assembly time if desired;
their conventions are described in detail in the following sections.
These options are specified simply by typing them as part of the
command line when calling ASMB, separated by spaces. Since there are
quite a number of options, it's possible to have the command line
exceed the line-length limit of the terminal being used. If this
is the case, a Control-E (^E) may be issued to provide a physical CR-LF
so that the command line may be continued. Note that a logical CR-LF
(same as typing RETURN an the console) terminates the command and
begins assembly. If the terminal being used automatically provides a
logical CR-LF at the physical end of a line, then a Control-E should
be issued before the end-of-line has been reached. The total line
length is limited to 128 characters by CDOS.

Options are specified only in the call to ASMB. The only exceptions
to this are the List Options (see below), which may be used in
slightly different form as operands of the LIST pseudo-op. Options
may be specified in any order; any number of the allowable options
may be specified at the same time. Consider the following sample
call of the file THISFILE.Z80 by ASMB:

          ASMB THISFILE RANGE PAGE=50 SYMB XREF OPCODE

Notice that the 3-letter drive-request instruction was not used in
this example; this means that all disk operations will involve the
current drive. The options specified ask ASMB to mark relative jumps,
format a listing page, and generate symbol and cross reference tables.
These are described in more detail below under the respective options.

Throughout this manual, the symbols "< >" are used to bracket quanti-
ties which are to be replaced by user-quantities, usually names of
files on the disk. However, in NO cases are the bracket symbols
themselves to be entered with the quantity involved. Also, throughout
this manual, the pseudo-ops are written in all-upper-case to set them
off. However, this does not mean that they must be written this way
in source-code. Both upper- and lower-case are acceptable.

++++++++++++++++++++++++++++++++++
Options Specified When Calling ASMB
++++++++++++++++++++++++++++++++++

============
List Options
============
The List Options are similar to the operands of the LIST pseudo-op
which may be part of a source file. However, the List Options as

specified at assembly time will override ANY and ALL LIST items given
in source code. Following are the four allowable List Options; they
are specified by typing the word(s) given here in the command line
calling ASMB. Note that if mistakenly both of a pair Cond-Nocond or
Gen-Nogen are specified, the one which appears last on the command line
has precedence. An important point to note is that NONE of the List
Options changes the actual object code assembled; they merely change
what is sent to the print-listing file on console or disk.

```
----
Cond
----
```
This option forces the generation and printing of all blocks of code
which are part of an IF definitions whether the IF is true or false.
The default is Cond if no LIST pseudo-ops are present in the source
file; therefore, it would generally be used as an option only to over-
ride all the Nocond operands of LIST in the source. Note that this has
NO effect on whether the IFs are satisfied or not.

```
---
Gen
---
```
This option forces the generation and printing of the Macro which
follows every Macro call. The default is Gen if no LIST pseudo-ops
are present in the source file; therefore, it would generally be
used as an option only to override all the Nogen operands of LIST in
the source.

```
------
Nocond
------
```
This option forces NO printing of IF or ENDIF statements and NO
printing of IF definitions (the code following the IF) if the IF
statement is false. In other words if the IF statement equals 0,
thus causing the code which is part of the IF not to be assembled,
the print-listing will likewise not contain the unused IF code.
If the IF statement has a value other than 0 and is thus true, the
print-listing will not contain the IF or ENDIF lines but will contain
the code of the IF definition; therefore, the included portion of code
will appear contiguously with the rest of the source code. The Nocond
option is used to override all the Cond operands of LIST pseudo-ops in
the source file.

```
-----
Nogen
-----
```
This option forces NO printing of the code following MACRO calls. How-
ever, NOTE that Macro DEFINITIONS are always printed as are the Macro
CALLS themselves; it is only the code which the Macros generate, the
Macro Expansions, which are not printed. This option prevents very long
print listings when using multiple Macro calls. Since the Macro code
will not be printed, neither will the object code which is printed on
the same line; however, Nogen does not in any way affect the object
code sent to the .REL file. The Nogen option is used to override all
the Gen operands of LIST pseudo-ops in the source file.

```
======================
Macro=<d:filename.ext>
======================
```
The Macro= option is one of the most powerful features of the Assem-
bler. It is used to specify the name of a disk file which is to be
searched to satisfy any Macros required at assembly time. During
Pass 1, the Assembler forms a Macro Definition Table (MDT) of the
Macros defined in a source program. (Remember that ABMB expects Macros
to be defined before they are called.) If the Macro= option is speci-
fied, a table is formed of the ADDRESSES of the Macros contained in
this library. Now, when an opcode is encountered in a source program,
the MDT is the first place searched to satisfy it. If it is not found
there, the Macro Address Table for the Macro Library (only when using
Macro=) is searched next; if the Macro is found, then the Macro Defini-
tion is loaded into memory from the disk. If the opcode is not found
in either of these places, the Opcode Definition Table (ODT) is
searched last. Thus, because ASMB searches for a Macro before an
opcode, it is possible to redefine Z-80 instructions using Macros (see

the chapter on Macros). Another advantage of this method of searching
is that the entire Macro Library specified does not become part of the
source code. Thus, you may specify a very large Macro-library file,
but only those Macros actually used are included in the assembled code.
Note that this is different from the INCLUDE pseudo-op in that the
INCLUDE would include the entire file, needed or not.

The Macro= command should be typed exactly as shown above, where the
user would insert following the "=", the filename.ext to be searched.
The "d:" represents the disk drive letter (one of A-D) and is optional;
if not specified, ASMB defaults to the current drive in its search for
the Macro file. The default for the Macro= option is, of course, no
Macro file searched; however, this does not in any way affect the
manner in which Macros intrinsic to the source code are handled.

```
======
Opcode
======
```
The Opcode option, when specified, will create a cross reference
listing of OPCODES and MACROS used, which will be sent to the console
or disk following the assembled-code listing. This cross reference
contains all of the opcodes used in the assembled program along
with the Macro names used, in alphabetical order, and the line
numbers of their definitions and places of occurrence. The first
column following the column of opcodes and labels is reserved for
the line numbers of the definition points of Macros, and is thus
blank for the opcodes. Subsequent entries contain the line numbers
of the places of occurrence of opcodes and Macros. Note that opcode
cross reference listings are limited in width by the Width option
and that a line will be stopped at the last complete entry which will
fit this width, and will be continued on the next line. The Opcode
option is very useful for debugging purposes as it allows you to find
all the occurrences of a particular opcode very quickly. The default
is no-Opcode cross reference table.

The Opcode Cross Reference is implemented by a disk sort. This means
that when this option is selected, ASMB creates a file on the CURRENT
drive called <filename>.$$0 where filename is the file being assembled.
Then, when assembly and the opcode cross reference are complete, this
file is deleted from the disk. Note that if the current drive does
not have room for the opcode temporary file, an error message is
printed and assembly is aborted (see also "write error" in the error
message chapter).


```
==============================
Page=<number decimal lines/page>
==============================
```
The Page option is used when generating a printer-listing to cause
the Assembler to calculate and display a specified number of lines
per page. At the top of each page ASMB will also print a heading,
a title if specified, and a page number. Note that even if several
lines are longer than the Width specification and wrap around, the
Page function will count these correctly, and will list the exact
number of lines specified per page (including the heading). The
default value is 60 and the limits are 10 to 254 lines. Note that
the Page=, Top=, and Width= options must be typed exactly as shown
(no spaces) in order to be interpreted correctly.

```
======
Parity
======
```
The Parity option is normally specified when assembling code which
was originally 8080 code and has been entered using Z80 mnemonics.
This is because the Z80 and 8080 microprocessors treat the parity
flag slightly differently and the Z80 may not execute 8080 parity
instructions correctly (the Z80 treats parity as an overflow flag
after arithmetic instructions). By specifying the Parity options
the user will be warned in the assembled listing of possible problems
along this line by the letter 'P' preceding the line numbers of the
affected lines. It is up to the user to determine whether or not
the parity flag is used correctly in a given situation. The instruc-
tions which will be marked are: JP PE,nn; JP PO,nn; CALL PE,nn;
CALL PO,nn; RET PE; and RET PO. The default is no-Parity.


```
=====
Range
=====
```
The Range option is used to have the Assembler tell you all those
places in code which currently use absolute jumps which are "within
range" for doing relative jumps. When specified, the line numbers
of the affected jumps will be preceded by the character "R". Thus,
the next time the code is edited for changes, the corresponding
absolute jumps may be replaced by relative jumps. Note that the
Assembler itself does NOT make the replacements. The default is
no-Range option.


```
==============
Symbol or Symb
==============
```

The Symb option is used to cause the Assembler to print the symbol
table following the listing. The symbol table lists all program or
data label names in alphabetical order from left to right in rows,
and the hex address which is the value of the label used by the Assem-
bler followed by the type of code segment there. For example the entry

          LABEL 00A7'

means that LABEL has the value 0OA7 in the relative-code program area.
(The symbols #, *, ", and ' are defined in the section on code
segments.) If the label belongs to an EXTernal, the address given is
that of its last OCCURRENCE in the present module, rather than its
actual value. Similarly, for a label defined by a DL (see Define
Label), the value listed in the symbol table is its last value to
occur in the source code. The Width of the symbol table in a printer-
listing will be the same as that of the code listing preceding it;
however, the line length of the symbol table will be limited to
include the last full label name and address which can be fit within
that width. The default is no-Symb table option.

```
==============================
Top=<no. dec. lines before top>
==============================
```
The Top option is used to specify the number of lines between the last
line of one page and the top or first line of the next page when

creating printer listings. This feature may be used to specify the
spacing between pages when creating listings. If the value 0 is
specified, formfeeds are issued to the printer at the end of each
page. This is the default value and is the one ordinarily used.
Notice that the values of Page+Top should equal the number of lines
desired per page of printed text. The limits are 0 to 255 lines.

```
==============================
Width=<number decimal columns>
==============================
```
The Width option is used to specify the number of characters of printed
text which will appear per line of a listing. This feature is used to
allow the use of different widths of paper in printer listings or to
allow for terminals capable of displaying different numbers of charac-
ters per line. The default value is 79, which should accomodate all
80-character terminals. The limits are 39 to 255 characters. If lines
longer than that specified are written, they will wrap around and be
continued on the next line of the listing. Note that the symbol table,
error listing, and opcode and label cross reference tables are also
limited by the Width specification.

```
====
Xref
====
```
The Xref option, when specified, will create a cross reference listing
which will be sent to the console, printer, or disk (as specified)
following the assembled-code listing. This cross reference contains
each of the label names used in the assembled program, the line

number of its definition, and the line numbers in numerical order
of each of its places of occurrence. The first column following the
column of labels is the column of line numbers of their definitions.
Note that if DL's (Define Labels) are used in the source code, those
label names may be defined more than once. Thus, in the cross re-
ference listing, the subsequent defining line numbers are preceded by
a '#' to set them off. Do NOT confuse this with the '#' for Data areas
defined elsewhere. The ENTRY pseudo-op will also generate a doubly-
defined label, once at the ENTRY point itself and once where the label
is actually defined. The Xref option is very useful for debugging
purposes as it provides an alphabetical listing of the locations
of every label used in a program. The default is no-Xref table.
Note that cross reference listings are limited in width by the Width
option and that a line will be limited to the last complete entry
which will fit within that specification; entries will then be
continued on succeeding lines.

The Xref Cross Reference is implemented by a disk sort. This means
that when this option is selected, ASMB creates a file on the CURRENT
drive called <filename>.$$$ where filename is the file being assembled.
Then, when assembly and the cross reference are complete, this file is
deleted from the disk. Note that if the current drive does not have
room for the cross reference temporary file, an error message is
printed and assembly is aborted (see also "write error" in Chapter 6).

++++++++++++++++++++++++++++++
Summary of Defaults and Limits
++++++++++++++++++++++++++++++

The default values and limits of the above options are summarized
here for convenient reference.

========
Defaults
========
In the absence of specified options, ASMB will default to these values:
no-Range, no-Parity, no-Xref, no-Symb, no Macro=, no-Opcode, Page=60
lines, Top=O (formfeed), Width=79 characters, default to List options
specified within source code as operands of the LIST pseudo-op.

======
Limits
======
The paper-managing options for generating printer listings are limited
to the following values: Page= 10 through 254 lines, Top= 0 (formfeed)
through 255 lines, Width= 39 to 255 characters. The lower limits on
Page and Width are imposed to assure that at least some code is printed
on each page.

****
CHAPTER 3: ASSEMBLER FIELDS

```
***************************
```

The Assembler recognizes four fields or different types of
expressions. These are: labels, opcode mnemonics, operands, and
remarks. The conventions which apply in the use of these four
fields are given below following remarks on the syntax of ASMB.
Any two of the four fields must be separated from each other by
at least one delimiter; these are: a tab, a space, a colon (after
labels only), a semi-colon (before remarks only), or a CR-LF (to
terminate lines). Multiple delimiters may be used to improve
readability.

```
-------------------------
```
Characters and Line Length
```
-------------------------
```
The Assembler accepts any printable ASCII characters in lines of
code. Specifically, this means any ASCII character having a hex
value between 20H and 7EH inclusive. In addition the three control-
characters, CTRL-I, CTRL-N, and CR are also recognized (^I is the
tab character which it translated into up to eight spaces by ASMB,
^N is the character to expand a line on the printer, and CR is a
carriage return). NO other control-characters are recognized by
ASMB. The maximum length of a line accepted by the Assembler is
80 characters, where the last character is the CR. Lines having
more than 80 characters will be truncated.

```
--------------------
```
Upper and Lower Case
```
--------------------
```
It would be good to mention at this point that the Assembler will
accept ALL commands, options, opcodes, pseudo-ops, filenames, or
any of the other Input it requires in both upper and lower case
or a combination of the two. This means that source code files may
be entirely lower case and will still be understood by ASMB. How-
ever, even though internally ASMB treats them the same, when listing
out the opcode and cross reference tables, because of the sort
routine used, there will be as many different entries as there are
variations in the label or opcode used. For example, the label
BEGIN will be a separate entry from the label Begin. This is actually
a useful feature; it is possible to have sections of code which use
the same data labels, but still have the ENTRIES in the cross reference
table remain separate. Thus, it is easier for the user to keep track
of the two sections while debugging. Note that the two labels are
ALWAYS equivalent to the Assembler.

```
+++++++++++++
```
Names (Labels)
```
+++++++++++++
```

Names are considered to be the labels of all instructions as well as
the operands of pseudo-ops such as ENTRY, EXT, and NAME. Labels may
be as long as desired (if all on one line); however, only up to the
first 6 characters are used by the assembler. Thus, the first six
characters of a label may not be duplicated in another label. The

first character of a label must be an alphabetic character or "."
or "$"; the remaining characters may be ".", "$", or any alphanumeric
(A-Z, a-z, 0-9). The delimiter for a label is generally a colon-space,
colon-tab, or a colon-CR-LF. However, the colon may be eliminated IF
the label begins in column one. Note that this means that opcode
mnemonics may NOT begin in column one. The operand may follow the
colon immediately if desired.

The following labels are illegal because the Assembler considers
them to be register names:

        A   B   C   D   E   F   H   L   I   R
        AF   BC   DE   HL   SP   IX   IY

These symbols are also illegal if written in lower-case.

++++++++++++++++
Opcode Mnemonics
++++++++++++++++

The ASMB Assembler recognizes all standard Z-80 mnemonics. For
the reader who does not have familiarity with these, they are well-
documented in the Z-80 CPU Technical Manual published by both Zilog
and Mostek. The following mnemonics are recognized by ASMB in BOTH
the forms shown. ASMB recognizes these opcodes in the form published
by Zilog and Mostek:
        ADC A,s; ADD A,n; ADD A,r; ADD A,(HL); ADD A,(IX+d);
        ADD A,(IY+d); SBC A,s; IN A,(n); OUT (n),A.
ASMB also recognizes them in this abbreviated form:
        ADC s; ADD n; ADD r; ADD (HL); ADD (IX+d); ADD (IY+d);
        SBC s; IN A,n; OUT n,A.
In addition the Assembler will allow either of the formats shown for
the following four instructions:
        IM   0   or   IM0
        IM   1   or   IM1
        IM   2   or   IM2
        DJNZ   nn   or   DJNZ,nn

Opcodes may begin on any column of a line EXCEPT column one. They
may be preceded by a label. They must be followed by a space or
tab as a delimiter between the opcode and the operands, or if there
are no operands and no remarks, the line is terminated by a CR-LF.

Pseudo-opcodes are a special form recognized only by the Assembler
and for which no object code is generated. The conventions of
ASMB for pseudo-ops are described in other sections. Some of
the common ones are ORG, EQU, EXT, ENTRY, DEFB, DEFS, DEFM, and END.

A special type of opcode is the MACRO name; when this is listed
in a column of source code, ASMB will insert the corresponding
code of the MACRO at assembly time. For more information on
this see the description of MACROs in Chapter 5.

```
++++++++
Operands
++++++++

Operands may consist of register names, constants, label names, or
expressions.  Register names include all standard Z-80 registers.
These are documented in the Z-80 CPU Technical Manual published by
Zilog and Mostek for the reader who is not familiar with their names
or purposes.  Constants consist of one of the five types outlined
in the Constants section below.  Names may include DATA labels,
program segment labels, subroutine names, COMmon names, EXTernals,
ENTRY names, EQUate statement labels, or the like; they must be
set up as described in the Names section above.  NOTE that names
of Macros may not be used as operands; instead, they are used
as opcodes and the assembler will substitute the correct code
at assembly time.  Also note that "operands" for statements such
as the TITLE and *INCLUDE statements are not operands in the
sense described here and are subject to other restrictions.

---------
Constants
---------
ASMB allows binary, octal, hexadecimal, decimal, and ASCII
constants according to the following conventions.

Binary - Numbers formed from binary digits (0,1) and terminated
by the character 'B'.  Range:
        -1111111111111111B<=n<=1111111111111111B.
Example:    LD    BC,10101101111010B

Octal - Numbers formed from octal digits (0-7) and terminated by
the character 'Q'.  Range:
                -177777Q<=n<=177777Q.
Example:    LD    BC,25572Q

Hex - Numbers formed from hexadecimal digits (0-9 and A-F) and
terminated by the character 'H'.  A hex number beginning with
a letter MUST be preceded by a '0' to distinguish it from a
label or register name.  Range:
                -0FFFFH<=n<=0FFFFH.
Example:    LD    BC,2B7AH

Decimal - Numbers formed from decimal digits (0-9) and EITHER
left unterminated or terminated by the character 'D'.  Range:
                -65535<=n<=65535.
Example:    LD    BC,11130

ASCII - Numbers represented by the ASCII character(s) itself
(themselves) enclosed in single quotes.  Range:
        ' ' through '~' which amounts to the values 20H through
        7EH, including all alphanumerics and punctuation.
Example:    LD    BC,'+z'
```

```
Note that each of the previous examples will produce the same value
```

in the BC register upon assembly and execution.

Current Program Counter - $

The "$" character may be used in the operand of any opcode allowing
expressions as operands. The "$" is used to represent the current
location counter of the Assembler. Note that "$" points to the BEGIN-
NING of the instruction which contains it and not to the end. An
example of the way to use it is:

        DATA: DB   0,11,3,2,7,24,17
        COUNT: EQU $-DATA

The name COUNT thus has the value of seven, because this is the number
of entries in DATA (the address of DATA subtracted from the current
location). Now elsewhere in the source program, the name COUNT can
be used to stand for the number of entries in DATA. There is great
advantage to this representation; if it becomes necessary to change
the number of entries of DATA and re-assemble, the value of COUNT
will be changed automatically. Whereas if an absolute 7 were used
instead of COUNT, every occurrence of the 7 in the source program
would have to be changed.

The "$" is often used in another way which is actually poor programming
practice, and that is to use the "$" in a relative jump instruction.
The best way to handle relative jumps is to label the location to be
jumped to, and use this label as the operand of the jump instruction.
ASMB will then calculate the correct displacement (see also "range
error" in Chapter 6). Remember that "$" represents the location coun-
ter at the start of the CURRENT instruction.

Expressions and Operators

The Assembler allows expressions to be used as operands, which it
evaluates at assembly time and places the calculated values in the
object code. These expressions may be used in place of either address
or constant operands, provided they do not evaluate to an illegal
quantity. The following operators may be used to form expressions.
Operators which are symbols (eg, "+") should NOT be separated from
their operands by a space. Operators written as one or more letters.
MUST be separated from their operands by a space. It is sometimes
desirable to group operations; however, parentheses could cause con-
fusion since they are also used for memory references. Therefore,
brackets ("[" and "]") are also acceptable to group the operands of
expressions. Parentheses may be used provided they do not begin an
expression or enclose one. Some examples will illustrate this; the
following are legal expressions, but they may be different from what
the programmer wished:

        LD   A,(X+Y)/Z
        LD   BC,(Q+R-S)

In the first example the "/Z" is ignored and the expression evaluates
to the contents of the address X+Y. The expression of the second
example means the contents of the address given by Q+R-S. These
examples may be rewritten slightly to change their meanings:

```
        LD      A,[X+Y]/Z
        LD      BC,[Q+R-S]
```

Now in the first example, the QUANTITY of X added to Y and divided by
Z is loaded into A, and not the CONTENTS of this address. In the
second example also, the-brackets mean QUANTITY, whereas parentheses
would mean CONTENTS. (Note that neither brackets nor parentheses are
required in this example.) An example in which either parentheses or
brackets may be used because because the meaning is not ambiguous is:

```
        ADD     A,Z/(X+Y)
```

The following lists the legal operators for expressions along with
an explanation of each:

```
        +          Addition or Plus - binary or unary
        -          Subtraction or Negative - binary or unary
        *          Multiplication
        /          Division
        MOD        Modulus - compute the remainder of a division
                   X MOD Y is defined to be X-(Y*INT(X/Y))
                      if X=23 and Y=7 then X MOD Y=2
        > or GT    Greater Than - true if the left operand is
                         greater than the right operand
        GE         Greater than or Equal - true if the left operand is
                         greater than or equal to the right operand
        < or LT    Less Than - true if the left operand is
                         less than the right operand
        LE         Less than or Equal - true if the left operand is
                         less than or equal to the right operand
        = or EG    Equals - true if the left and right
                         operands are equal
        NE         Not Equal - true if the left and right
                         operands are not equal
        SHL n      Shift Left Logical - shift n places
                         if X=2AH then X SHL 1=54H
        SHR n      Shift RIGHT Logical - shift n places
                         if X=2AH then X SHR 2=0AH
        NOT        Logical Not - unary
        AND        Logical And -
                      if X=C0H and Y=47H then X AND Y=40H
        OR         Logical Or -
                      if X=C0H and Y=47H then X OR Y=C7H
        XOR        Exclusive Or -
                      if X=C0H and Y=47H then X XOR Y=87H
```

ASMB considers these operators to have a hierarchy that determines
which take precedence over others. The list which follows gives this
hierarchy, progressing downward from those of highest priority to
those of lowest priority; all those operations on any given line are

of equal priority. Thus, operators which are on the same line of the
hierarchy would be evaluated from left to right as they occur in an
expression. However, operators or parts of expressions enclosed in
parentheses or brackets are evaluated first, beginning with the inner-
most set. The hierarchy is:

```
*, /, MOD, SHL, SHR
+, -    (unary)
+, -    (binary)
NOT     (unary)
AND
OR,XOR
>, <, =, GT, LT, EQ, NE, LE, GE
```

All operations not marked are assumed to be binary. If the result of
an expression is 0, the expression is false; if the result of an ex-
pression is other than 0 (specifically -1), the expression is true.
Also two operands which are equal result in a true expression; two
that are not equal result in a false expression. This information is
important for determining how to satisfy the IF operand. See the
chapter on Conditional Assembly and Macros for more information on the
IF statement. Also, see the chapter on Error Messages (specifically
"Expression Error") for information on which of the above operators may
be used with labels belonging to relative (REL) program segments.

+++++++
Remarks
+++++++

The remarks field is free-format including any printable ASCII
characters as long as the comment is preceded by a ';'. The
remark may follow an opcode, operand, or label or may exist on
a line by itself. The ';' may be in column one if it is desired
to have the remark on a line by itself. Multiple blanks or tabs
may be used before or within the remark to improve readability. A
CR-LF terminates the remark. Remarks may appear on any line, ie, fol-
lowing any of the legal opcodes or pseudo-ops except TITLE and FORM.

```
*******************************************************
```
CHAPTER 4: PSEUDO-OPCODES RECOGNIZED BY THE ASSEMBLER
```
*******************************************************
```

The following section contains an alphabetical list of the pseudo-ops
recognized by ASMB. They are all listed here for convenient reference;
however, several-of the pseudo-ops are described in other sections.
Certain of the pseudo-ops require labels; others require no label.
More information on this may be found under "missing label" and "label
not allowed" in the chapter on error messages. Macros and Conditional
Assembly are explained in detail in a separate chapter.

```
+++++++++++++++++++++++++++++++
Alphabetical List of Pseudo-ops
+++++++++++++++++++++++++++++++


===========================
ABS (Absolute code segment)
===========================
```
The ABS pseudo-op is described in the Source Code Segments section at
the end of this chapter.

```
========================
COM (COMmon code segment)
========================
```
The COM pseudo-op is described in the Source Code Segments section at
the end of this chapter.

```
=======================
DATA (Data code segment)
=======================
```
The DATA pseudo-op is described in the Source Code Segments section at
the end of this chapter.

```
=======================
DB or DEFB (Define Byte)
=======================
```
The DB pseudo-op is used to tell the Assembler to reserve a byte or
string of bytes as data in the object code. The bytes may be speci-
fied using any of the forms of constants described in the Constants
section of Chapter 3, or as a series of labels which have been
previously defined or EQUated to a value. NOTE that if the value
of the label or constant exceeds the range 0 to FFH (or its equivalent
representation in decimal, octal, or binary), the DB will generate an
expression error and insert a null. Also note that either of the
terms DB or DEFB may be used. The format of the DB pseudo-op is

          <Label:> DB <Item or List of Items>

where the label is optional and the item or list is any of: a byte,
a string of bytes separated by commas, a string of ASCII characters,
or an expression or string of expressions following the rules for
expressions outlined in Chapter 3 (note that the expression must be
equivalent to an absolute byte). The length of the string of bytes is
limited by the length of a line for ASMB (80 total characters). A
string of ASCII characters must be enclosed in single quotes. If it
is desired to represent the single quote itself in a string, it must
be given as two adjacent single quotes (''). Some examples will
illustrate the use of DB:

          DB   'how are you?'   (the string will be converted
                                           to ASCII bytes and stored
                                           in consecutive memory loca-
                                           tions in the object code)
```

```
        DB      -2,-4,-6,10,11,17          (in order the hex bytes
                                            which will be stored
                                            are:  FE,FC,FA,0A,0B,11)
```

=========================
DL or DEFL (Define Label)
=========================
The DL pseudo-op is similar to the EQUate statement and is used to
define the value of a label. The major difference between DL and EQU
is that DL can be used to set a label to different values at different
times in the assembly of a particular program. The format of DL is

```
        <Label:>  DL   <Expression>
```

where both the label and the expression are required. The expression
may be in the form of another label or an arithmetic expression which
is a combination of names or constants and which follows the conven-
tions for expressions outlined in Chapter 3. However, note that the
expression can NOT be a string of bytes, nor can the expression use
any EXTernal names. The DL command is exactly like the SET pseudo-op
of some other assemblers. An example of its use follows:

```
        START:  LD      SP,...
                :
                :
        COUNT:  DL      4
                LD      A,COUNT
                :
                :
        COUNT:  DL      COUNT-1
                LD      B,COUNT
                :
                :
                END     START
```

In this example COUNT is redefined later in the source program from
its original value. Note that only the original definition of COUNT
need be changed for both of them to be changed upon re-assembly.

It's important to note here that the DL command is quite unlike the DB,
DM, DS, and DW commands although their formats all similar. These
other commands all cause the Assembler to reserve a specified number
of bytes in the object code, whereas DL is an Assembler DIRECTIVE but
does NOT reserve any bytes. The DL statement is used to define a value
or values internally to ASMB.

===========================
DM or DEFM (Define Message)
===========================
The DM pseudo-op is exactly similar to the DB pseudo-op except that
the DM command sets the high bit (Bit 7) of the last byte in the string
of bytes following the command, when this string is converted to object
code. This is a very convenient feature for defining ASCII strings
(in which Bit 7 is not used), provided the user-program tests this

bit to determine the end of a string. Note that the DB command leaves
the high bit of the last byte unchanged. The format of DM is

        <Label:> DM <Item or List of Items>

where the label is optional and the item or list is any of: a byte,
a string of bytes, a string of ASCII characters, or any expression
or string of expressions following the rules for expressions outlined
in Chapter 3 (expression must evaluate to be 8-bit absolute, however).
As was the case for DB, a string of ASCII characters must be enclosed
in single quotes ('). If it is desired to represent the single quote
itself in a string, it must be given as two adjacent single quotes.
An example of the use of DM is

                :
        CR: EQU   0DH
        LF: EQU   0AH
                :
        STRING: DM   'this is a string',CR,LF
                :

In this example the last byte of the string (LF or 0AH) would be
placed in the object code as 8AH with the high bit set. Note that
the length of a DM command is limited to the 80-character total line
length of ASMB. Allowing for a space in column 1, the characters
"DM", a space, the opening "'", and the CR at the end of the line; this
means that the maximum length of a single string using the DM command
is 74 characters. However, preceding DB statements may be used to
accommodate longer strings.

===========================
DS or DEFS (Define Storage)
===========================
The DS pseudo-op is used to tell the Assembler to reserve a specified
number of bytes in the object code for storage. Note that ASMB will
not insert any particular values in these reserved bytes. The format
of the DS command is

        <Label:> DS <Expression>

where the label is optional and the expression is either a constant
or an expression which evaluates to an absolute and which follows
the rules for expressions outlined in Chapter 3. Please note that all
the terms of the expression used MUST have been previously defined in
the source code or an error will result. A constant value of 1 causes
ASMB to reserve one byte. An example of DS is

        ADDRSTABL: DS   20

in which 20 bytes are reserved by a program to be used as an Address
Table of 10 entries (two bytes per entry). Note that either of the
terms DS or DEFS is allowed by the Assembler.

========================

```
DW or DEFW (Define Word)
========================
The DW pseudo-op is used to tell the Assembler to reserve a word or
string of words in the object code.  A word is defined to be 2 bytes.
Thus, the DW pseudo-op might be used to specify a look-up table of
absolute addresses.  The words may be specified using any of the forms
of constants described in the Constants section above, or a label which
has been previously defined or EQUated to a word.  Note that either of
the terms DW or DEFW is recognized by ASMB.  Also note that the Assem-
bler places the low byte FIRST, treating every word of two bytes as
though it were an address.  For example, the word "0C923H" would appear
in the object file as the two bytes, "23H" followed by "C9H".  Likewise,
if LABEL1 had been previously defined as "0C923H", a "DW LABEL1" would
generate the same two bytes, "23H" followed by "C9H".  This follows the
conventions described elsewhere for expressions or labels used as ope-
rands anywhere in the source code.  In general the DW pseudo-op is
associated with addresses and the DB statement with data; however, this
is by no means an absolute.  The DW pseudo-op is a very convenient way
for entering addresses because the user does not need to keep track of
placing the low byte before the high byte; simply enter an address as
it is written.  The format of DW is

        <Label:>  DW  <Item or List of Items>

where the label is optional and the item or list is any of:  a word,
a string of words, or an expression or string of expressions following
the rules for expressions outlined in Chapter 3 and evaluating to an
absolute word or string of words.  The length of the string of words
is limited to the 80-character total line length expected by ASMB.
However, successive DW commands may be given to accommodate longer
tables of words.

Unlike the DB statement, an expression which exceeds the legal range
for a DW will not cause an "expression error".  Instead, the expression
will be evaluated modulus 65,536.  See "value error" in the chapter on
error messages for a further explanation of this.  Note, however, that
with the DW statement ASCII character strings longer than two bytes are
not allowed.  Some examples will illustrate these ideas:

        DW      'AA'            (evaluates to 4141H)
        DW      'A'             (evaluates to 0041H)
```

```
        DW      'ABC'           (illegal, expression longer
                                    than one word)
        DW      100H,1ACH,-814  (multiple expressions, evaluates
                                    to the hex bytes, in order.
                                    00,01,AC,01,D2,FC)


====================
END (End of assembly)
====================
The END pseudo-op is used to terminate assembly of a block of source
code.  The format of END is
```

```
          <Label:>   END   <Expression>
```

where the label is optional and the expression is subject to these
rules: ONLY the main module of a program should have an expression or
name following, and this module MUST have this expression. The expres-
sion should be equivalent to the entry point of the module at which
execution will begin. All other modules are then terminated with the
END statement alone and are thus considered by ASMB to be sub-modules.
The reason for this convention is that the Linker/Loader must know in
which of the modules and at what address to begin execution. The
quantity in the Expression may contain any legal operators (see section
on Expressions in Chapter 3). Following is a sample use of the END
statement to terminate assembly of a main module:

```
            ENTRY   MAIN
      MAIN: LD      SP,1800H
            :
            :
            END     MAIN
```

whereas this example shows termination of a sub-module to be linked
to the main module:

```
            EXT     MAIN
      BEGIN: LD     A,1O
            :
            :
            END
```

The END command is a signal to the Assembler that a logical body of
code is complete. Therefore, only one END statement should appear
in a module. Should the END appear in the middle of a block of code,
everything following the statement will be ignored by ASMB.

```
============================
```
ENDIF (END of IF definition)
```
============================
```
The ENDIF pseudo-op is used to terminate Conditional Assembly of a
block of code which follows an IF statement. The formats of IF and
ENDIF are described in detail in the following chapter on Macro and
Conditional Assembly.

```
====================================
```
ENTRY (Entry point for these modules)
```
====================================
```
A program module may be assembled with unresolved addresses providing
they are declared EXTernal in that module. Any address declared
EXTernal to one module must be declared an ENTRY in another module.
These two modules are eventually linked. Since these addresses are
unresolved, they are represented in the EXT and ENTRY statements as
label names. The names then become a part of the .REL file. The
Linker/Loader reads the .REL files at run time, determines the unre-
solved addresses, and places their correct values in those bytes which
expect the addresses. If the Linker is unable to resolve an address,

it prints the undefined label name an the console followed by an
asterisk (see Part II for more information on LINK).

The ENTRY pseudo-op is used to declare in a source-file that that file
contains the entry point(s) of the listed names. These names may be
label names of subroutines, or program or data blocks. The format of
the ENTRY command is

            (no label) ENTRY <Namel,Name2,...>

The number of names used as operands of the ENTRY pseudo-op is limited
only by the total line length (80 characters). Extra names in ENTRY
not actually defined in the source-file will produce the error message
"undefined symbol". ENTRYs may appear anywhere within a program
module, but are typically written at the top of a file to be easily
seen in the print-listing. ENTRY labels (standing for corresponding
addresses) can be referenced by any other module which declares those
names to be EXTernal (see EXT section). Refer to Part II on the Linker
for information on linking these modules at run time.

Below is an example of a module which uses an ENTRY statement to
demark two subroutines and a table of data:

                    ENTRY   METRIC,ENGLIS,CONTBL
            METRIC: ...   ; metric-to-English conversions
                    :
                    RET
            ENGLIS: ...   ; English-to-metric conversions
                    :
                    RET
            CONTBL: ...   ; conversions table
                    END

The corresponding example of a program module which calls these sub-
routines is given with the description of EXTernals.

============
EQU (Equate)
============
The EQU pseudo-op is used to inform the Assembler that two named
quantities are equivalent. The format of EQU is

            <Label:> EQU <Item>

where the label is required and the item is any of: a constant, an
address, a label, or an expression following the rules given in
Chapter 3. Note that all the terms of the expression MUST have been
previously defined. Also, the expression may NOT involve the names
of any EXTernals.

The EQU statement is used to equate a label to a particular value.
Once this label is defined, it is defined for the entire source
program. The DEFL command should be used for labels which are to
change within a module. EQU is a useful statement for simplifying or

clarifying source code. For example suppose the ASCII characters for
carriage return (CR) and line feed (LF) were to be used throughout a
source program. Instead of using their values, a clearer procedure
would be to enter the lines:

```
        CR:     EQU     0DH
        LF:     EOU     0AH
```

somewhere in the source program and then use the names "CR" and "LF"
to stand for the values as in:

```
        STRING: DB      'end of text',CR,LF
```

The EQU statement is also very valuable for changing a quantity quickly
and in all places. Suppose that it is desired to test a program with
different values for a timer. Suppose further that this value is used
10 times throughout the source code. If the original value is used in
each of those 10 places, then all 10 will have to be changed to change
the timer. However, if each of the 10 places uses the label "TIMER"
and the following statement appears somewhere in the module:

```
        TIMER:  EQU     <value>
```

then this statement can very easily be changed by editing. This as-
sures upon re-assembly that all the places TIMER occurs will be changed.

```
=====================================
EXT or EXTRN (these modules External)
=====================================
```

Using ASMB, program modules may be assembled with unresolved addresses.
The EXT pseudo-op is used to declare in a source-file that that file
must depend on some other module(s) to satisfy certain EXTernal names.
The EXT and corresponding ENTRY names become parts of the two .REL
files; the addresses are then resolved at run time. Further informa-
tion about this is described in the first paragraph under the ENTRY
pseudo-op; information about linking and running files is given in
Part II on the Linker. The format of the EXT command is

```
        (no label)  EXT  <Name1,Name2,...>
```

where the names may be label names of subroutines, or program or data
blocks. Note that Module Names under the NAME pseudo-op may NOT be
used in the EXT fields of other modules. The number of names used as
operands in an EXT pseudo-op is limited only by the total line length

of ASMB (80 characters). Either of the forms EXT or EXTRN is accepted
by the Assembler. EXTs may appear anywhere within a program module,
but are typically written at the top of a file to be easily seen in the
print-listing. When the assembled modules are linked and run, all
EXTs must be satisfied by corresponding ENTRYs in other modules or the
Linker will return an error message.

It is important to note that label names declared EXTernal to a module
may be used as operands within the module, but may NOT be used in

expressions. For example the following lines would be legal:

```
        EXT     COUNT
        :
        :
        LD      A,COUNT
        :
```

whereas the following would be illegal and would generate an error:

```
        EXT     COUNT
        :
        :
        LD      A,COUNT+3
```

Also note that a label name declared as an EXTernal to a module may
NOT be redefined (ie, used in the label field) within that module.

Below is an example of a module which uses an EXTernal statement to
declare the names of two outside subroutines and a table of data:

```
            EXT     METRIC,ENGLIS,CONTBL
    START:  ...
            :
            LD      HL,CONTBL
            LD      A,(HL)
            CALL    METRIC
            :
            INC     HL
            LD      A,(HL)
            CALL    ENGLIS
            END     START
```

The corresponding example of the module which contains these subrou-
tines is given with the description of ENTRY pseudo-ops.

```
=====================
FORM (paper Formfeed)
=====================
```
The FORM command is used to advance the paper in a print listing to
the top of the next page. The format of FORM is

```
        (no label)  FORM  (no operands)
```

FORM is used for clarity in a print-listing, as the beginning of a
routine can be more clearly identified if it starts at the top of a
page. The FORM command in the source code will not be printed on the
listing. Multiple FORM commands may also be used; however, each page
will be numbered and titled by ASMB. The command "EJECT" may be used
in exactly the same way as FORM to force a paper-feed to the top of
the next page.

The Assembler implements FORM by issuing a series of linefeeds to the
printer; the number of linefeeds needed to reach the next page is

determined by ASMB from the TOP= and PAGE= options. The only exception
to this is that when TOP=0 has been-selected, an actual formfeed
character is issued to the printer. In this case the printer will
advance according to the paper size for which it is designed.


```
==============================
IF (begin Conditional Assembly)
==============================
```
The IF pseudo-op is described in the following chapter on Macro and
Conditional Assembly.


```
=====================================
*INCLUDE (Include the given disk file)
=====================================
```
The *INCLUDE pseudo-op is used to specify a source-file on disk which
is to be included in the assembly of the present source. The format
of the *INCLUDE command is

```
        *INCLUDE <d:filename.ext>
```

where d stands for the disk drive letter (A-D), and filename is the
user's disk filename, which may or may not include a 3-letter extension.
If the disk drive letter is omitted, ASMB assumes the file is on the
current drive. It is IMPORTANT to note that the *INCLUDE statement
MUST begin with the asterisk in column one. Hence, NO label field is
permitted with this opcode. The filename may follow the *INCLUDE after
at least one delimiter (space or tab). Another important point to
notice about the *INCLUDE is that all of the given file is included in
the present file. Hence, if the included file has an END statement,
this END statement will terminate assembly of the present source when
it is encountered. An example will illustrate this; suppose this is
the source-file to be assembled:

```
        BEGIN:  LD       SP,...
                  :
                  :
        *INCLUDE A:USERFILE.Z80
                LD       HL,...
                LDIR
                END
```

and suppose the following is USERFILE.Z80:

```
        START:  LD       BC,...
                LD       DE,...
                END
```

Because the USERFILE contains an END statements the Assembler will
never see the LD and LDIR instructions of the source-file. Assembly
will be terminated following the inclusion of USERFILE. To avoid this
problem simply leave off the END statements of files which are to be
INCLUDED in the assembly of other files, or put the *INCLUDE statement
as the.last one in a source program and leave off that source's END
statement.

The *INCLUDE statement is particularly useful in conjunction with
Conditional Assembly blocks of code (see the discussion of the IF
statement in the next chapter). For example a file may be INCLUDEd
depending on whether or not an IF statement is satisfied. Also, the IF
statement can be used to determine which of several files will be
INCLUDEd. An example of this use of *INCLUDE follows; one of three
different files will be included and the others ignored dependent on
the value of the label DECIDE (defined earlier in the source):

```
                :
                :
                IF       DECIDE EQ 0
        *INCLUDE A:MOVROUTN.Z80
                ENDIF
                IF       DECIDE EQ 1
        *INCLUDE B:SAVROUTN.Z80
                ENDIF
                IF       DECIDE EQ 2
        *INCLUDE LOADROUT.Z80
                ENDIF
                :
                :
```

where the first file would be found on drive A, the second on drive B,
and the third on the current drive. The entire block of code above
could be put in a file of its own called DECIDFIL. The user source
file would initialize the variable DECIDE using an EQU or DL statement
and give the command "*INCLUDE DECIDFIL". Then, based on the value of
DECIDE, the routine which would be included would be either the move,
save, or load routine above.

*INCLUDEs may be nested up to four levels; more than this will generate
a nesting error. The example above illustrates two levels of nesting:
the user file INCLUDEs the file DECIDFIL, which in turn INCLUDEs one
of the files MOVROUTN, SAVROUTN, or LOADROUT. It is also possible to
write a source program which consists of *INCLUDEs only.

```
===================================================
LIST (use following commands to generate Listings)
===================================================
```
The LIST pseudo-op is used to set the Assembler print-listing options.
This at NO TIME affects the actual object code put out by ASMB. It
is simply used to suppress undesired or repetitive sections of the
listing file. The format of the LIST statement is

```
        (no label)  LIST  <Option1,Option2,...>
```

where the options are taken from the list of six legal operands which
follows this paragraph. The number of options which may be placed on
a line is limited only by the line length. However, 3 options is the
practical limit because more than this will result in duplicate or
conflicting options. Options may be given in any order. If conflic-
ting options are given (conflicting options are the pairs gen-nogen,

cond-nocond, on-off), only the last one of the pair on the line will
be used.

The LIST command may be used as often as desired throughout a source-
code file. However, note that if the List Options of Chapter 2 are
issued at the time of the CALL of ASMB, these will override any corres-
ponding LIST commands given in the SOURCE. For example if the List
Option "Gen" is specified when calling ASMB, all "Nogen" operands of
LIST in the source would be overridden. However, the OTHER operands of
LIST in the source would still be effective. Following are the six
allowable operands of the LIST pseudo-op. Information on the use of
List Options when calling ASMB will be found in Chapter 2.

OFF (turn Off assembly listing)

Suppress print-listing until end of code or an ON option. This option
is de-selected when assembly of a program begins (before encountering
a LIST pseudo-op).

ON (turn On assembly listing)

List print-listing to disk-file or console until end of code or an OFF
option. ON is the default when assembly of a source file begins.

COND (begin listing Conditional Assemblies)

Force the generation and printing of all blocks of code which are parts
of IF definitions, until end of code or a NOCOND option. COND is the
default when assembly of a source file begins; therefore, it would
generally be expressed only to override a previous NOCOND option. Note
that COND forces the printing of IF statements but NOT the assembly of
them; that is determined by whether the IF statement is true or false.

GEN (begin listing Generated Macros)

Force the printing of the Macro expansion following every Macro
call, until end of code or a NOGEN option. GEN is the default when
assembly of a source file begins; therefore, it would generally be
selected only to override a previous NOGEN option.

NOCOND (do Not print Conditional Assemblies)

Force no printing of IF or ENDIF statements and no printing of IF
definitions (the code following the IF) if the IF statement is false.

In other words an IF definition which is not assembled due to its
being false will not be listed in the print-listing either. This
option will remain selected until the end of code or a COND option.
The option is de-selected when assembly of a program begins and thus

must be first selected using the LIST pseudo-op. Selection of NOCOND
in NO WAY affects the object code of an assembled file.

```
--------------------------------------
NOGEN (do Not print Generated Macros)
--------------------------------------
```

Force no printing of Macro expansions. However, note that Macro
definitions are always printed as are the Macro calls themselves; it
is only the code which the Macro generates which is not printed.
This option will remain selected until the end of code or a GEN option.
The option is de-selected when assembly of a program begins and thus
must be first selected using the LIST pseudo-op. Selection of NOGEN
in NO WAY affects the object code of generated macros of an assembled
source file.

```
==============================
MACRO (begin Macro definition)
==============================
```

The MACRO pseudo-op is described in the following chapter on Macro and
Conditional Assembly.

```
===========================
MEND (Macro definition End)
===========================
```

The MEND pseudo-op is used to terminate the block of code which forms
a Macro Definition. The formats of MACRO Definitions and Calls, and
the MEND statement are described in detail in the following chapter.

```
==================
NAME (module Name)
==================
```

The NAME pseudo-op is used to assign a name to a particular module
for use by the Linker. This name is, however, written in alphanumerics
so it is also useful to the programmer for remembering the purpose of
the module. The format of NAME is

          (no label) NAME <Module Name> (1-6 characters)

where the module name should follow the same syntax rules as for
labels. The NAME statement is optional; it is not required for linking
of modules. However, if the NAME statement is omitted, the Assembler
automatically assigns the first six characters of the filename to be
the module name. Note that NAME is different from TITLE. The TITLE
statement merely tells ASMB to print a heading at the top of each
page of the listing but has no effect on the object code; NAME forces
the name of the module to be saved as part of the .REL file. Thus,
a library manager program is able to locate .REL files by name.

Page 33

```
============
ORG (Origin)
============
```

The ORG pseudo-op sets the Assembler location counter and is used when
it is desired to start assembly of a block of code at a particular
address. This location may be set by the user to be absolute, or it
may be left up to the Assembler to determine the value of the ORG. The

location counter may be set to a value as often as desired in a source.
program; that is, multiple ORG statements may be used. The format
of the ORG is

        <Label:> ORG <ABS Address, Label Name, or Expression>

where the label is optional but the expression or address is required.
The VALUE of the ORG (ie, the address of the first statement following
the ORG) is determined by the value of the label or expression. Note
that all the terms used in the expression MUST have been previously
defined. The TYPE of code segment which will follow the ORG is
determined by the type of code segment to which the label or expression
belongs. For example, the statement "ORG LEFTOFF" would continue
a COMmon area if LEFTOFF belonged to a COMmon area, and would continue
a RELocatable area if LEFTOFF belonged to a RELocatable program area.
In either case, however, the value of the ORG would be determined by
the value of LEFTOFF. The statement "ORG 100H" would begin an
ABSolute program area since the address is absolute. Note that the
ORG pseudo-op does not reserve any bytes, but merely specifies an
address at which those bytes are to begin.

==============================
REL (Relocatable code segment)
==============================
The REL pseudo-op is described in the Source Code Segments section at
the end of this chapter.

===================================
REM (Remark beginning in column one)
===================================
The REM statement is another method of designating a remark; however,
REM assures that the remark is always printed beginning in column 1 of
a print-listing without any characters preceding (as with the ";").
The REM pseudo-op itself is never printed as is also the case with the
FORM and TITLE printer-controls. The format of REM is simply

      (no label) REM <Remark Phrase> (as many char. as will fit line)

Some printers (for example the CROMEMCO 3700-series Printers) will
expand a line if the line contains the Control-N (0EH) character.
This is the reason for the REM statements to be able to give this
character at the beginning of a remark and have the printer expand the
line to make it more noticeable. However, when using the ^N feature,
the user must take care that the remark to be printed does not exceed
HALF the width specification of the Width= option. For example most
listings use the default value of Width=79; thus, the number of char-
acters in the REM statement which uses the ^N should not exceed 39.
This is to prevent the printer from printing off the side of the paper.
Also note that the maximum length of a REMark is 74 characters.

===
TITLE (Title to be printed at top of each page)
===
The TITLE pseudo-op is used to print a title at the top of each page

a print-listing beginning in column 1. The format is simply

    (no label) TITLE <Title Phrase> (as many char. as fit on line)

As with the REM statement, the Title Phrase may contain the character
Control-N (0EH). On CROMEMCO 3700-series printers this character will
expand the line to twice its normal width. For this reason when using
the ^N in a TITLE statement, the number of characters in the Title
Phrase should not exceed half the number of characters which will be
specified in the Width= option. The TITLE command should be the first
line of a program in order to be printed on Page 1 as well as the
other pages. Note that titles may be changed in the middle of a source
program simply by giving a new TITLE command. Also note that in such
a case TITLE causes an automatic FORM feed. The maximum length of a
Title Phrase is 72 characters; strings longer than 72 are truncated.
The Assembler inserts a blank line where the Title Phrase would be if
TITLE is not specified.

++++++++++++++++++++
Source Code Segments
++++++++++++++++++++

Perhaps the single most important feature of the CROMEMCO Assembler is
its ability to generate relocatable code. This feature allows a user
to assemble a number of modules of source code separately, and link
them together in any order at run time. It also means that the object
code can be executed at nearly any address in memory (it is generally
not advised to assemble and run programs over portions of CDOS). The
Assembler can assemble all data in locations separate from the program
area so that either area may be programmed into ROM.

There are four special pseudo-ops which inform the Assembler what type
of object code to generate. This section describes these four Source
Code Segment pseudo-ops and the ways they are used. Explanations of
the way relocatability works are given in the following, and informa-
tion on the CROMEMCO Linker/Loader may be found in Part II.

========
ABSolute
========
The ABS pseudo-op precedes a code segment which is to be assembled in
absolute code (absolute addresses). The Linker will consider this code
to be non-relocatable. The format of ABS is simply

        (no label) ABS (no operand)

All the code following the ABS will be considered to be absolute until
another Code Segment pseudo-op is given. The Assembler defaults to REL
upon start of assembly of a program; thus, if no pseudo-op is given,
the object code will be relocatable. ABS areas are addressed conti-
guously throughout a source program unless ASMB is told otherwise by
the use of ORG statements. At the beginning of a program the program
counter for ABS is set to 0 (however, unless ABS is specified, ASMB
assumes the REL pseudo-op) unless the user overrides it with an ORG.
For subsequent ABS areas the current contents of the program counter

(which is the address at which the last ABS left off) will specify
the loading address. Some examples will help illustrate these ideas.
Consider the following:

```
            ABS
    START:  LD      SP,...
            :
            :
            END
```

In this example the entire source program is to be considered absolute
and the addresses are to begin at 0. If the same example were written:

```
            ORG     1000H
    START:  LD      SP,...
            :
            :
            END
```

the entire source program is again to be considered absolute with the
addresses beginning at 1000H. Now consider two ABS areas and a DATA
scratch-pad area between them:

```
            ABS
    START:  LD      SP,...
            :
            :
    STOP:   LD      HL,...
            DATA
    ADDR1:  DS      2
    ADDR2:  DS      2
            :
            ABS
    NEXT:   LD      BC,...
            :
            :
            END
```

In this example assembly will begin at absolute location 0 because no
ORG statement is specified. Assembly of the second ABS area will begin
with the next address following the LD instruction at STOP. Note that
an ORG statement could have replaced either ABS statement to cause the
code segment following to assemble elsewhere. The DATA area will be
assembled relocatable.

======
COMmon
======
The COM pseudo-op precedes a data segment which is to be assembled
common to more than one module. The name and size of the COMmon(s)
are then saved in the .REL file; this enables the Linker to load the
addresses correctly at run time such that the given area is common
to several program modules. The CROMEMCO Linker/Loader is the same
one used to link .REL files produced by the CROMEMCD FORTRAN IV

Compiler. Hence, COMmons are a very convenient way of enabling a
machine language subroutine to use a FORTRAN data area, or as a fast
way to pass arguments between FORTRAN and machine language programs.
COMmons may be used in this way for assembly language programs as
well; two or more program modules may use the same data scratch-pad
area for passing arguments. EXT and ENTRY statements which apply to
data areas may be replaced by COMmons. (When interfacing FORTRAN to
machine language routines, allow four bytes for Real, two bytes for
Integer, and one byte for Logical variables.) The format of COM is

        (no label) COM <Common Name> (1-6 characters)

Note that the full word COMMON is NOT allowed by ASMB. The Common Name
may be omitted in the above, and this is considered the blank common.
If the name is used, it should follow the rules for labels given in
Chapter 3. Following the COM command are the labels and pseudo-ops
allocating storage. Note that when COMmons are used by more than one
program module, they must either be the same length in every module,
or the module which is linked first must contain the longest COMmon
specification so that LINK allocates at least that number of bytes.

Also, note that the COMmons of different modules DO NOT have to have
the same labels on the data. Thus, this COMmon in one module:

                COM   DATA
        ADDRTB: DS   20
        COMMTB: DS   10

and the following COMmon in another module:

                COM   DATA
        COUNT: DS   4
        LOOKUP: DS   26

would assemble and link correctly; they are the same length and the
data labels are transparent to the Linker.

There are 15 different COMmons of equal level (ie, there is no
hierarchy) allowed by the Assembler in any one program; exceeding
this number will generate an error. All the code following the COM
will be considered to be a common until another Program Segment
pseudo-op is given. A common may also be continued later in the
same program segment by giving the COM command with the same name as
before. Using a different name will cause the COM location counter
for THAT common to start over at zero. Remember that COMmons of the
same name need not be the same length in every module as long as the
module containing the longest COMmon specification is linked first.
An example will illustrate some of the features of COM:

        BEGIN: LD   SP,...
                :
                :
                COM   INSTRC
        TABLE1: DS   50

```
            REL
            LD        HL,...
            :
            :
            COM       ADDRES
   LOCATE:  DS        20
            COM       INSTRC
   TABLE2:  DS        50
            END
```

Both TABLE1 and LOCATE in the above will begin at COM location-counter
zero; however, note that they are different commons. TABLE2 will begin
at location counter 50 for COM INSTRC (thus COM INSTRC reserves 100
total bytes as storage). Also note the use of the REL statement to
return to RELocatable code following the end of the first part of COM
INSTRC. Generally the DS pseudo-op is used to allocate storage area
for a COMmon; if the DB, DM, or DW statements are used, bear in mind
that the loaded bytes of the first COMmon may be over-written by the
second loaded COMmon when they are linked.

====
DATA
====
The DATA pseudo-op precedes a program segment which is to be assembled
as a block of data. LINK will consider this code to be rolocatable.
The format of DATA is simply

        (no label) DATA (no operand)

All the code following the DATA will be considered to be part of the
data block until another Code Segment pseudo-op is given. The DATA
pseudo-op is very similar to REL; DATA is provided so that the user
may maintain separate data and program code segments in a source file.
Thus, the program segments may be programmed into ROM following their
being linked and loaded, and the data segments may remain in RAM, for
example. All DATA segments of a program are based upon the DATA
location counter, which is set to zero upon the start of assembly.
As is the case with ABS and REL, all DATA segments in a program will
be addressed contiguously if ORG statements are not used to change the
addresssing. Also, remember that an ORG will cause assembly to
continue with the type of Code Segment to which the expression of
the ORG statement belongs. For example the following section of a
source code program:

```
            DATA
   LABEL1:  DS        10H
            REL
            LD        A,...
            :
            :
            DATA
   LABEL2:  DS        10H
            END
```

would be assembled in exactly the same way as this section:

```
            DATA
LABEL1: DS      10H
            REL
            LD      A,...
            :
            :
            ORG     LABEL1+16
LABEL2: DS      10H
            END
```

where the ORG statement has replaced the second DATA statement. Since
LABEL1 belongs to a DATA area, the ORG statement tells ASMB to return
to assembling in the DATA code segment without the need for the second
DATA pseudo-op. For more information on the ORG see the list of
pseudo-ops above.

===========
RELocatable
===========
The REL pseudo-op precedes a code segment which is to be assembled in
RELocatable form. The Linker will recognize this code at run time,
link it with any other relocatable modules, and load them into the
desired address in memory. Relocatability works in this way: following
assembly, the .REL file contains the locations of all bytes which
contain unresolved addresses. At run time the Linker then determines
the place at which the program is to be run and correctly fills in
the unresolved addresses. As the modules are linked, LINK also prints
the names of any still undefined labels (those declared in EXT
statements).

The Assembler defaults to the REL code segment upon start of assembly
of a program and the REL location counter is set to zero. However,
other code segment pseudo-ops may be specified throughout the source,
and REL issued to return to relocatable code at the end of these
segments. The format of REL is simply

        (no label) REL (no operand)

The code following the REL will be considered to be relocatable program
area until another Code Segment pseudo-op is given. REL areas are
addressed contiguously throughout a source program unless ASMB is told
otherwise by the use of ORG statements. Note, however, that the ORG
will cause assembly to continue with the type of Code Segment to which
the expression of the ORG statement belongs (see the example of this
in the section above on the DATA pseudo-op). The REL statement is
generally needed only to return to relocatable program code following
the use of another Code Segment opcode. Note also that data may
be included in the REL area. The DATA and REL pseudo-ops treat
relocatable code in an identical manner; therefore, unless there is
a specific reason for keeping the data and program areas separate,
the DATA statement(s) could be eliminated.

```
******************************************
CHAPTER 5:  MACRO AND CONDITIONAL ASSEMBLY
******************************************
```

Two of the most powerful features of the CROMEMCO Relocatable Assembler
are Macro and Conditional Assembly. The purpose of this chapter is to
define and explain these two features and illustrate their use with
examples. However, the user should bear in mind that these examples
only scratch the surface as illustrations of the uses of Macros. It
is left up to the readers to adapt Macro and Conditional Assembly to
their needs.


```
+++++++++++++++++++++++++++++++++++++++++++
Macro Assembly (MACRO definition and calls)
+++++++++++++++++++++++++++++++++++++++++++
```

Macros provide the user with a method of producing a block of in-line
code in a source file without having to generate this block of code
each time it is required. This block of code is known as the Macro body.
Macros also allow a great deal more flexibility than in-line source
code because of the ability to accept parameters. This means the Macro
may be tailored to suit a particular purpose. For example suppose a
user wishes to use a move routine which does a block move of 100 bytes.
Later in the same program, a block move of 500 bytes is desired. Al-
though these two routines could be written separately, it would be much
easier to write a Macro which accepts the correct parameters and
generates the correct block move. Some other advantages of the use
of Macros are:

        -Rewriting repetitive blocks of code is not required. The
           code is written only once in the Macro.
        -Macros can be used to improve program readability and to
           create easily-read skeleton programs.
        -Macros written by a number of programmers can be collected
           in a Macro library which may be used by all. Eventually
           nearly entire programs may be written using the Macros in
           this library.
        -New Z-80 instructions may be designed using existing instruc-
           tions in a Macro (this is an instruction only to the
           Assembler; it is not possible to add instructions to
           the Z-80 Instruction Set).
        -An error found in a Macro need be corrected only once
           regardless of the number of times the Macro is called.

Some users may wonder how Macros differ from subroutines, since
subroutines may also be used to reduce the coding of frequently
executed blocks of code. One distinction between the two is that
subroutines branch to another part of the program while Macros
generate in-line code. However, a Macro does not necessarily

generate the same source code each time it is called. The source

code the Macro generates can be changed by changing the parameters
in the Macro Call. Also, Macro parameters can be tested at
assembly-time by the Conditional Assembly (IF) statement. These
two features enable a general purpose Macro Definition to generate
customized source code for a particular situation. Thus, the
biggest difference between Macros and subroutines is that Macro
expansion and customized code result at assembly-time within the
object code. Subroutines, on the other hand, reside in the source
program, and require extra execution time (especially if the
subroutines do any conditional operations). There is a trade-off,
however, between the extra memory required for Macros (in-line code)
and the longer execution time of subroutines. In most cases using
a single subroutine rather than multiple in-line Macros will reduce
the overall program size. However, the use of Macros may be more
efficient in situations involving a large number of parameters.
Note that Macros can call subroutines, and subroutines can contain
Macro Calls.

An example of a simple Macro Definition would perhaps illustrate some
of the afore-mentioned points. Suppose that there were a number of
times in a source program that it was desired to exchange the upper
four and the lower four bits of the A register. Although a subroutine
could be written to do this, the associated CALLs and RETurns would
slow down execution time. Thus, to save typing when writing the
source code, a Macro is used:

```
        ROTATE: MACRO
                RLCA
                RLCA
                RLCA
                RLCA
                MEND
```

The general format of a Macro Definition can be seen from this example.
The word ROTATE becomes the Macro name. Thus, to CALL this Macro one
would simply use the word ROTATE as an opcode in the source code, and
the Assembler would insert the four RLCA opcodes as in-line source code
following the ROTATE Macro opcode. This is known as the Macro EXPANSION.
The MEND statement informs the Assembler that the Macro Definition is
complete. Suppose now that rather than be limited to having the Macro
exchange the high and low bits of the A register only, it was desired
to have it operate on any of the 8-bit registers. The following Macro
Definition might be used in place of the above:

```
        ROTATE: MACRO     #REGIS
                RLC       #REGIS
                RLC       #REGIS
                RLC       #REGIS
                RLC       #REGIS
                MEND
```

This Macro uses the parameter REGIS, the value of which it will
determine when the ROTATE macro is called. The "#" symbol is
required to precede the parameter(s) everywhere it appears in a Macro

Definition to distinguish it from other labels; however, this symbol
is NOT required when specifying the parameter in a Macro Call. Since
ROTATE now expects one parameter, the form of a Call would be:

                 ROTATE   <register>

where the word "register" would be replaced with one of: A, B, C, D,
E, H, or L. The Assembler would then generate in-line code using
the correct register name. For example if the Macro Call "ROTATE H"
was used, ASMB would generate the in-line code:

                 RLC   H
                 RLC   H
                 RLC   H
                 RLC   H

Based on the above examples we now give the complete format of a Macro
Definition and Call. A Macro is defined by:

        <Macro Name:> MACRO <#Parameterl,#Parameter2,...>
                       <opcodes and
                         operands which
                         may use the
                         parameters of
                         the Macro statement
                         and which form
                         the Macro body>
        (no label)   MEND   (no operand)

where the parameters are optional and are limited in number only by
the length of a line for ASMB (80 characters). The Macro Name is
required and is the name used when calling a Macro. The MEND is
the Macro End statement and is required to inform the Assembler
that the source code of the Macro is complete. The opcodes or
pseudo-ops between the MACRO and MEND statements comprise the
Macro Definition, and may be any legal Z-80 instructions, calls
to other Macros, or ASMB pseudo-ops.

There are a number of important points to note about the above
format of Macros. First, note that when passing parameters to
the Macro the parameter name must be preceded by the symbol "#"
everywhere it appears in the Definition; however, it is NOT used
to precede parameters in a Macro Call. The parameters are actually
dummy names; they stand for a quantity which will be substituted
at assembly time. Therefore, the same parameter name may be used
in several separate Macro Definitions (for example #REGIS may be used
more than once). The parameters MUST follow the syntax rules for
whatever portion of code they represent. Note that the text itself
of the actual (not dummy) parameter is substituted in the Macro
Expansion. Thus, register names can be used rather than a value
which stands for the register as in some other assemblers; see
the above example where the letter H is used as the parameter.
Another way this is useful is to substitute for letters in the
opcode itself:

```
          ROTATE: MACRO   #DIR,#REGIS
                  R#DIRC  #REGIS
                  R#DIRC  #REGIS
                  R#DIRC  #REGIS
                  R#DIRC  #REGIS
                  MEND
```

In this example either the command RLC or RRC could be generated by
assigning the letter "R" or "L" to the first parameter. However,
if the letter "Q" was used, this would generate the illegal opcode
"RQC', causing an error message when the Macro is expanded. A last
point to be made about parameters in Macros is that parameter names
that appear early in a list should NOT be subsets of parameters that
fall later in a list. This is because dummy parameter names do not
have a delimiter (such as a colon) to inform the Assembler of their
last character; note that parameter names do not follow the same
syntax rules as label names. Dummy parameter names may be as many
characters as will fit on the line and be composed of any printable
ASCII characters. An example of an illegal use of parameters is:

```
          LOAD:   MACRO   #OPER,#OPERND
```

where the user desired one parameter to be the operation and the other
the operand. This is illegal as it stands because OPER is a subset
of OPERND. A correct example is:

```
          LOAD:   MACRO   #OPERAT,#OPER
```

Another important point to be made about the format of Macro
Definitions concerns the way in which labels are defined. Labels
appearing in DL statements within the Macro Definition are not
subject to the following restriction because they can be multiply-
defined (see section on Conditional Assembly for an example of the
use of a DL and an IF statement to cause conditional Macro assembly).
A label appearing on any other statement of a Macro Definition will
generate a multiple definition error if that Macro is called more
than once (the second expansion would also reproduce the label).
To avoid this problem a general label name for Macros has been
provided, which is used by assigning two letters to the label name
followed by the characters "#SYM". These four characters are replaced
by a four-digit number each time a Macro is called. The four-digit
number starts at 0000 and is incremented by one each time ANY Macro
is called, whether or not it is the given Macro. Thus, for example
the dummy label name AA#SYM in this Macro:

```
          BITEST: MACRO   ...
                  :
                  :
          AA#SYM: LD      HL,...
                  :
                  JP      AA#SYM
```

would be assigned the actual label name AA0000 if BITEST was the first
Macro called in the program. The next Macro call would increment this
to AA0001, and the next to AA0002, etc. In general do NOT use #SYM

as the name of a parameter in a Macro Definition; the effect of this
is that the current value of #SYM will be used instead of the desired
parameter.

The final point to be made concerning the format of the Macro
Definition concerns nesting of Macros. Macro Definitions may be
nested indefinitely; this means there can exist a Macro Definition
which completely contains a Macro Definition which completely
contains a Macro Definition, and so on indefinitely. However, Macro
Calls may be nested to eight levels maximum. This means there can
exist a Macro Definition which contains a Macro Call, whose Macro
Definition contains a Macro Call, whose Macro Definition contains
a Macro Call, and so on up to eight levels deep. Exceeding this
limit will generate a nesting error. Note that a Macro may also call
itself, provided there is a Conditional way (see IF) of ending the
self-calling before the ninth level. An example of nested Macro CALLS
will be found in the examples section later in this chapter.

Some special notes are necessary on nested Macro DEFINITIONS. The
Assembler does not evaluate a Macro Definition within a larger,
outside Macro Definition until the larger definition is called.
This means that the outside Macro should be called BEFORE the inside
Macro to avoid generating a phase error. The benefit of nesting
Macro Definitions may not be obvious; the following example illus,-
trates one level of nesting used to define several different Macros:

```
        DEFINE: MACRO    #1,#2
        EX#1#2: MACRO
                PUSH     #1
                PUSH     #2
                POP      #1
                POP      #2
                MEND
                MEND
```

This nested definition may then be called in a source program as follows:

```
                :
                DEFINE   BC,HL
        START:  ...
                :
                EXBCHL
                :
```

The opcode "EXBCHL" was defined by the call to DEFINE; other calls to
DEFINE could define such source code segments as "EXAFBC" or "EXBCDE".
After the initial call to DEFINE the necessary PUSHes and POPs to
generate a double register exchange will be inserted into the source
code by the call "EXBCHL" used as an opcode. The DEFINE Macro could
be resident in a Macro Library to further save typing. Note, however,
that DEFINE must be called once for every Macro which it defines and
that this call must precede the call to the nested Macro.

The above functions could also have been implemented by the single
Macro:

```
        EXCH:   MACRO   #1,#2
                PUSH    #1
                PUSH    #2
                POP     #1
                POP     #2
                MEND
```

The difference here is that the parameters must be specified each
time the Macro is called. For example a Call in a program would be:

```
                :
                EXCH    BC,HL
                :
```

Either of the above examples could be used to create a Macro to
exchange register pairs. Note the differences between them. There
is a more advanced example of nested Macro Definitions in the last
section of this chapter.

The above sections describing the details of a Macro Definition are
provided for reference. However, a better feeling for the ways in
which Macros may be used will come after these details are illus-
trated by means of examples. The last section of this chapter
provides examples of the uses and correct formats of Macro
Definitions and Calls. The last thing to be described in this
section is the format of the Macro Call:

```
        <Label:>  <Macro Name>  <Parameterl,Parameter2,...>
```

The label is optional; the parameters are also optional if none are
specified in the Macro Definition. That is, the parameters in the
Macro Call must match those in the Macro Definition in number and
order; they are NOT, however, preceded by the "#" symbol (because
these are the actual, not the dummy parameters). The Macro Name
should match the name appearing in the label field of the MACRO
statement. At assembly time the Macro will be expanded and the
source code generated will be printed on consecutive lines following
the Macro Call statement (unless NOGEN is selected--see List Options
and LIST pseudo-op). Each of these lines will have a plus, "+",
sign immediately following the line number of the print-listing to
distinguish these lines as belonging to a Macro Expansion. Note that
Macro Call statements may appear anywhere throughout a source program
including within another Macro Definition (beware of nesting to more
than eight levels deep, however).

An important point about Macro Calls and Definitions is that a Macro
must be defined in a source program BEFORE it is called. This is to
prevent a phase error from occurring. The general practice is to give
all Macro Definitions near the beginning of the source code, followed
by the body of the program itself. One of the most interesting
features of the CROMEMCO Relocatable Assembler is that Z-80

instructions can be redefined (in terms of other Z-80 instructions)
using Macros. Of course, such an instruction which is redefined
can not be used in its traditional sense again within the same
source program; however, there are specialized cases in which it
is desirable to slightly modify the function of an instruction.
Note that the instruction itself cannot be modified; it is merely
redefined in terms of other Z-80 instructions.

The way ASMB interprets instructions is an important part of
understanding the Macro capability. The Assembler forms a Macro
Definition Table (MDT) of the Macros residing in the source program.
This is the first place searched to satisfy an opcode. The second
place searched is a table of addresses specifying the Macros which
are accessed by the source program and which reside on disk (this
table is formed ONLY if the Macro= option is specified when calling
ASMB). If an opcode is found in this address tables the required
Macro Definition is read into memory from the disk and added to
the MDT. Finally, any still unsatisfied opcodes are found in the
Z-80 Opcode Definition Table (ODT). Thus, it is possible to write
an entire source program consisting only of Macros. In expanding
these Macros, ASMB then uses the ODT to evaluate the Z-80
instructions. This feature means that ASMB may be used as a
language compiler by having a library of Macros which translate
the commands of the language into a series of Z-80 instructions.
To avoid wasting memory and repeating Macros unnecessarily when
using such a scheme, Conditional Assembly may be used in conjunction
with Macros to automatically generate subroutine calls. This feature,
along with the other features of Conditional (IF) Assembly, are
described in the following section. At the end of this chapter is
a section of examples illustrating some of the features described
in these first two parts of the chapter.

++++++++++++++++++++++++++++++++++++
Conditional Assembly (IF statements)
++++++++++++++++++++++++++++++++++++

An often close associate of the Macro is the Conditional Assembly or IF
statement. The IF statement allows the user to write a source program
in which certain blocks of code are assembled or not depending on the
satisfaction of particular conditions. This is especially useful in
conjunction with the MACRO or *INCLUDE statements. When using the IF
statement with *INCLUDE, particular files may be included or not depen-
ding on values in the source program. Note that such a file may be a
series of Macros which are needed in the source program only under
certain conditions. The IF statement is useful with MACRO definitions
as a means of determining the desired number of levels of nesting of a
Macro within itself (this is illustrated in an example in the following
section). The feature may also be used to cause a Macro to set up a
subroutine the first time the Macro is called, and to generate a
subroutine CALL upon subsequent Macro calls. The format of the IF
statement is as follows:

```
     (no label)  IF  <Item>
                     <opcodes and
                       operands which
                       form the hody
                       of code to be
                       assembled
                       conditionally>
     (no label)  ENDIF  (no operand)
```

The item following the IF may be any legal label names expression,
or constant as described in Chapter 3. It will be evaluated by the
Assembler to determine whether it is True or False; a False expression
is one that evaluates to 0, and a True expression is one that evaluates
to -1 (0FFFFH). However, ANY non-zero value is considered to be True.
NOTE that the IF statement evaluates the expression as a sixteen-bit
quantity. If the expression exceeds this limit (for example: '0000'
is a 32-bit (4-byte) ASCII expression the correct expression is:
0000 or simply 0), it will generate an error message. A constant
which exceeds the range will, however, be evaluated MOD 65,536 and
will generate no error. Note that the Expression in the IF statement
may use the operators described in Chapter 3. All the terms of the
expression MUST have been previously defined to avoid errors; also,
the expression must evaluate to an absolute quantity. An example of
an IF statement with an expression is:

```
          IF      COUNT EQ 0
```

This will generate a value of True (or -1) if COUNT is equal to 0.
The example could have been written a different way:

```
     COUNT:  DL      1
             :
             :
             IF      COUNT
```

which will generate a value of True because in this case COUNT has the

value of 1 which also stands for True (non-zero). Note the difference
between these two examples; in the first case COUNT must equal 0 for
the expression to be True, and in the second case COUNT must equal
anything but zero for the expression to be True.

After evaluating the expression the Assembler will then assemble
the code following the IF statement if and only if the expression
evaluated to be True. If the expression was False, the block of code
bounded by the IF and ENDIF statements will simply be ignored by ASMB.
It is also possible to suppress the print-listing of such ignored code
by using either the NOCOND List Option or the LIST NOCOND pseudo-op
(see the appropriate sections for more information). An ENDIF
statement is required for every IF statement in a source program
to tell the Assembler when Conditional Assembly is finished.

IF statements may be nested up to eight levels deep; more than this will
generate an error message. IF statements may also be nested in Macros;

this makes it possible for a Macro to call itself a number of times
specified by the IF statement (an example of this may be found in the
following section). Macro parameters may be used in the expression of
the IF statement. The following example to do three rotates illustrates
this:

```
ROTAT3: MACRO    #DIREC
        IF       '#DIREC' EQ 'R'
        RRCA
        RRCA
        RRCA
        ENDIF
        IF       '#DIREC' EQ 'L'
        RLCA
        RLCA
        RLCA
        ENDIF
        MEND
```

Note that the actual ASCII value of the parameter may be specified by
enclosing it in single quotes as with any ASCII string. The two IF
statements check to see if the parameter specified when calling ROTAT3
is "R" or "L"; if it is neither, then no source code is assembled. If
one or the other, then the corresponding left or right rotates will be
generated.

Page 49

+++
Examples of Macro and Conditional Assembly
+++

Many of the features of MACRO and IF statements described above are
made clearest by illustrating them by means of examples. This section
is included to give the user some idea of the many ways which Macro
and Conditional Assembly may be used.

============================
Example 1: Block Move Macro
============================

The Macro Definition which follows provides a fairly simple example
of the use of a Macro. This Macro defines a method for easily
generating a block-move of a portion of a program:

```
MOVE:   MACRO    #SOURCE,#SRCEND,#DESTIN
        LD       HL,#SOURCE
        LD       DE,#DESTIN
        LD       BC,#SRCEND-#SOURCE
        LDIR
        JP       #DESTIN
        MEND
```

Note that three parameters are expected: a starting and ending location
for the source, and a destination; this is of the same format as the
M (move) command of DEBUG. Thus, the Macro Call for this example
might be part of a program such as:

```
                ORG       2000H
        LOAD:   MOVE      START,STOP,100H
        START:  LD        ...
                :
                :
        STOP:   END       LOAD
```

In this example the program would begin execution with LOAD, and would
move the block of code between START and STOP to absolute address 100H.

==
Example 2: A Macro that Converts Itself into a Subroutine
==

In some cases the in-line coding which results from many Macro Calls
is undesirable due to memory requirements. In such a case a Macro
can be created which converts itself to a subroutine. Such a Macro
has both the advantages of a Macro and a subroutine. Following is
the Definition for SUBMAC, a Macro which calls itself:

```
        TRUE:   EQU       -1
        FALSE:  EQU       0
        FIRST:  DL        TRUE
        SUBMAC: MACRO
                IF        NOT FIRST
                CALL      SUBROT
                ENDIF
                IF        FIRST
        FIRST:  DL        FALSE
                JP        DONE    ; causes program to jump around
        SUBROT: ...               ; subroutine upon first call
                :
                :
                RET
        DONE:   NOP               ; program jumps here
                ENDIF
                MEND
```

The first three lines above are not part of the Macro Definition,
but the value of FIRST must be initialized before it is used in the
Definition. The JP DONE instruction in the above is used to cause
a jump around the subroutine when it is assembled in-line with the
source upon the first Call to SUBMAC. A sample program which might
use this Macro is:

```
        START:  ...
                :
                SUBMAC
                :
                SUBMAC
                :
                END       START
```

The first Call to SUBMAC above would generate the subroutine itself

in-line. After the first call the value of FIRST has been redefined
to be FALSE; hence, the second Call to SUBMAC would generate simply
the line: CALL SUBROT.

```
====================================================================
Example 3:  Nested Macro Definitions to Generate Rotate Instructions
====================================================================
```

A number of interesting and useful functions can be implemented by
using nested Macro Definitions or Calls. The following is one such
example, making use of one level of nested Macro Definitions to
define a number of different Macros:

```
        ROTATE: MACRO     #SHFT
        M#SHFT: MACRO     #NUM,#REG
        VALUE:  DL        #NUM-1
                #SHFT     #REG
                IF        VALUE NE 0
                M#SHFT    VALUE,#REG
                ENDIF
                MEND
                MEND
```

The Macro ROTATE may be used to define a number of shift and rotate
Macros; however, the inner Macros are not defined until ROTATE has
been called one time. Thus, at the beginning of the program in which
we wish to use the Macros, it is necessary.to initialize them by
the Calls:

```
        SETUP:  ROTATE   SRA
                ROTATE   RRC
                ROTATE   RR
                ROTATE   SRL
                ROTATE   RLC
                ROTATE   RL
                ROTATE   SLA
```

Note that this will define 7 additional Macros with the names MSRA
through MSLA. The M (or any other legal character) is necessary
in order to avoid having the Macro names match Z-80 opcodes. (Note
that these same Z-80 opcodes are used within the Macro Definitions.)
We can now call any of these Macros, giving a number and a register
as parameters:

```
        START:  LD        ...
                MSRA      4,A
                MRLC      8,B
                MRR       3,E
                END       START
```

The number in each of the above cases is the number of shifts or
rotates which will be generated. Thus, the Macro Call "MSRA 4,A"
will, when expanded, generate 4 "SRA A" instructions in the source
code. Since the ROTATE Macro could be contained in a Macro Library,

the user's source program could contain a Macro Call of this type.

```
************************************
CHAPTER 6:  ASSEMBLER ERROR MESSAGES
************************************
```

The Assembler generates a number of error messages while assembling
to inform you of its progress. These messages fall into two general
classes: those that involve the actual call to ASMB, are generated
shortly thereafter, and are sent to the console; and those that are
generated while the source code is being assembled and which inform
the user of incorrect structures in the code. These two classes are
described below. The user should note that in most cases the Assembler,
when encountering an error, will assemble the line such that the correct
number of bytes are reserved. Thus, the addresses are still numbered
correctly, and the program may be loaded into memory and the incorrect
bytes changed using DEBUG. This saves reassembling a very long program,
when the user plans to debug it anyway. Of course, in the final
version of source code, the error should be corrected.


```
++++++++++++++++++++++++++++++++++++++++++++++++++
Error Messages Generated Following a Call to ASMB
++++++++++++++++++++++++++++++++++++++++++++++++++
```

The following list contains the error messages set off in two lines
of dashes, followed by a brief description of their meanings. Note
that the errors are printed exactly as they would be sent to the
console, upper or lower case. All the errors described in this
section will ABORT the assembly and return control to CDOS. The user
should be aware that any temporary files created by ASMB will remain
on the disk following an abort; these may be erased if desired, but
this is not required if the error is fixed and the file reassembled.


```
---------------------
source file not found
---------------------
```
This is generated when ASMB cannot find the specified source file on
the disk. Check your spelling of the filename and the disk directory
for the file.


```
------------------
no directory space
------------------
```
This is generated when ASMB attempts to open an output file (.REL or
.PRN, for example) and finds that there are already 64 entries (the
maximum allowed by CDOS) on the disk. This is NOT the same as running
out of disk space (see following error message). There may be 64
directory entries which are all short files, and thus not all the
available kilobytes may be used (81 Kbytes for small and 241 Kbytes
for large disks).

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

write error, file - <filename.ext>

ASMB will open any files which it requires (.REL or .PRN files, or the
temporary files it opens to manage the XREF and OPCODE listings),
shortly after being called. This message is generated if the disk is
full (81 Kbytes for small, 241 Kbytes for large) when these files are
opened, or if a file being written to causes the disk to become full
during assembly.

Note: The temporary files for XREF and OPCODE listings are created
only if one or both of these options are specified. The XREF file
is named <filename>.$$$ and the OPCODE file is named <filename>.$$0,
where <filename> is the one specified in calling ASMB. These files
are created during Pass 1 of the Assembler; they are removed from the
disk following completion of the assembly.

----
selected disk error

This message is generated if the 3-letter drive-request instruction
given after the filename is incorrect, ie, if it specifies a drive
which does not exist or is not one of the characters, "X" or "Z".
An example which will generate this message is: ASMB TESTFILE.ABE.
"E" is not a correct drive letter.

----
invalid option

This message is generated if an invalid or misspelled Option is speci-
fied following the <filename> in a call to ASMB. This will also appear
if an invalid delimiter (such as ",") is used between Options. One
or more <space(s)> is the only valid delimiter to separate Options.

MACRO library not found

This is generated only if the Macro=<d:filename.ext> option has been
issued and the filename cannot be found by the Assembler on the speci-
fied drive.

----
out of memory

The Assembler program (ASMB.COM) is loaded into memory at 100H and
begins execution there. Above itself in memory ASMB forms the symbol
table, which grows upward. Above the symbol table but below CDOS
ASMB forms the Macro Definition Table (MDT), which grows downward
through memory. If a user-program being assembled contains a great
number of Macros and/or symbols, the symbol table and the MDT may
grow together, thus generating the "out of memory" error. The message
will be printed on the console and assembly will be aborted at this
point. The simplest solution to the problem if it occurs is to edit
the source code into two or more separate modules, assemble them
separately, and link them at run time.

```
+++++++++++++++++++++++++++++++++++++++++
Error Messages Generated During Assembly
+++++++++++++++++++++++++++++++++++++++++
```

The following list contains the error messages generated during assem-
bly of source code. They inform the user of a wide range of incorrect
specifications such as misspelled opcodes or invalid relative jumps.
When an error occurs, ASMB prints the error message which applies on
the line immediately following the error. The message is a complete
expression, not a symbol, and it occupies the entire line in a print-
listing. It is set off by being preceded and succeeded by a string of
asterisks. If the print-listing is sent to the disk or is not generated
at all, any errors occurring during assembly will still be printed on
the console; in this case the entire line of code as generated in the
listing along with the error-type will be printed. Following assembly
the total number of errors will be printed. Also, at the end of the
listing will be printed a summary of all the line numbers where errors
have occurred during assembly. This summary is printed (either to the
disk or to the console) in the form of a table; the Width= option will
limit the length of lines of characters in this table, but will not end
any line in the middle of an entry just as was the case with the cross
reference tables. Note that for each type of error message up to 100
entries will be printed in this table. The error summary table is a
very useful feature for going back and editing the file for corrections.
Below in alphabetical order are the error messages which may appear
along with a brief explanation of each one.

```
--------------
argument error
--------------
```
This arises when an invalid constant is used. This might happen when
a number is incorrect for its base, or when an ASCII character string
is too long for an expression. For example, the lines

```
        LD      A,108Q          (8 not valid octal character)
        LD      HL,'ABC'        (too many ASCII chracters)
```

will both generate argument errors.

```
-------------------
divide by zero error
-------------------
```
This arises when an evaluated expression involves an attempt to divide
by zero. An example is

```
        END:    EQU     0FFF8H
                :
                :
                LD      HL,255/(END+8)
```

Since the value of END+8 is 0, this would produce the divide by zero
error.

Page 55


```
----------------
expression error
```

This applies to operand expressions which involve certain illegal
operations with labels belonging to REL, DATA, or COM code segments.
Expressions involving relocatable labels are limited to the following
operations:

```
        RELNAM+ABSNAM           (relocatable)
        RELNAM-ABSNAM           (relocatable)
        RELNAM-RELNAM           (absolute)
```

where RELNAM stands for a label belonging to a relocatable code segment
and ABSNAM stands for a label belonging to an ABS code segment (see REL
and ABS in the chapter an pseudo-ops). The type of expression of
the result of the given operation is given to the right in parentheses.
Also note that in the last case above both RELNAMs must belong to the
same type of Code Segment or an error will also be generated. (For
example a label belonging to a COMmon area may not be subtracted from
a label belonging to a REL area.) An expression error is generated
if any arithmetic is attempted (ie, an expression is formed) using
EXTernal names, as the values of these are unknown to the Assembler.
This does not mean that EXTernals may not be used as operands, of
course. Relative jumps from one type of Code Segment to another will
also generate expression errors; for example it is illegal to jump
from a REL to a DATA area using a relative jump. The reader should
refer to the section of Chapter 3 on operators for more information
on the use of expressions.

----
file not found

This message is printed following an INCLUDE for which the file to be
included cannot be found on the disk. This error does not terminate
assembly, but further errors may be generated if the source code looks
for labels belonging to the missing file. Note the difference between
this message, which is printed in the listing, and the "source file
not found" and "MACRO library not found" messages which abort assembly
and are printed on the console.

----
label error

This arises when a label contains or begins with an illegal character.
The characters 0-9 are legal within a label but are illegal as the
FIRST character. Allowable characters for labels are A-Z, a-z, ".",
and "$". All register names are also illegal as labels; these are
listed in Chapter 3 under the section on labels.

----
label not allowed

The following list of pseudo-ops do not ALLOW labels to precede them
because of their nature. This message is printed if a label is used
before one of the following pseudo-ops:

```
       ABS            FORM or        EXT or         MEND
       COM            EJECT          EXTRN          NAME
       DATA           ENDIF          IF             REM
       REL            ENTRY          LIST           TITLE
```

Note that this is NOT the error message which is printed in the case
of an illegal character in a label (see "label error"). Although the
"label not allowed" message is printed and counted as an error, the
source code will still be assembled correctly and the incorrect label
will be ignored by ASMB.

----
missing label

This message is printed when the following pseudo-ops are NOT preceded
by a label: EQU and DEFL or DL. This is opposite to the above case
(see "label not allowed"); note that these two pseudo-ops REQUIRE a
label to be assembled correctly. A MACRO definition section of code
also requires a label in order to be used by the Assembler. For more
information an Macros see the chapter devoted to them.

----
multiple definition

This message occurs any time a data or program label is defined more
than once. This is prone to happen when using INCLUDES, as the included
file may contain a label also used in the source file. Simply re-edit
one of the files and change the label(s) involved.

----
multiple MACRO definition

This error is exactly similar to the "multiple definition" above, but
is caused by a multiply-defined Macro name.

----
nesting error

This message appears whenever INCLUDEs, MACROs, or IFs are nested
beyond the levels allowed by the Assembler. These are: 8 levels of
nesting for MACROs and IFs, and 4 levels of nesting for INCLUDEs.
Note that this means 8 levels of nesting for Macro CALLS; Macro
DEFINITIONS may be nested indefinitely. However, be sure, when
nesting Macro definitions, to insert the correct number of MEND
(Macro END) pseudo-ops; the Assembler might otherwise consider a
portion of the Source code to be part of a Macro. Examples of both
nested Macro calls and definitions appear in the chapter on Macros.

----
no matching IF

This message appears following an ENDIF pseudo-op which has no corres-
ponding IF statement in the source code preceding it.

```
no matching MACRO
-----------------
This message appears following a MEND pseudo-op which has no corres-
ponding MACRO definition statement in the source code preceding it.


------------
opcode error
------------
This follows an opcode which is illegal; this may be because it is
misspelled or because the user intended for it to be a Macro and
forgot to include this Macro definition.


-----------
phase error
-----------
This error message follows a line containing a name which was defined
differently between Pass 1 and Pass 2 of ASMB.  The most common cause
of this is that a label has been used as a value (such as in an EQUate
statement) before it has been defined.  An example is


        LABEL1: EQU     LABEL2
                 :
                 :
        LABEL2: LD      A,5


LABEL2 has been used in the EQU statement before it was defined.  The
error is corrected by moving the offending statement (in thit case the
EQU statement) to follow the label definition.  Other causes of phase
errors are (1) using a term in an expression in a DS or IF statement
which has not been defined yet, and (2) calling-a Macro before it has
been defined.


-----------
range error
-----------
This message follows a relative jump which exceeds the range allowed
for such jumps.  This range is -126 bytes to +129 bytes measured from
the address at which the relative jump is located; the actual values
generated by the Assembler are in the range -128 to 127 because the
Z-80 measures relative jumps from the instruction following the jump.

The Assembler requires an ADDRESS, usually specified by means of a
label, to be used as the operand of a relative jump instruction.  ASMB
then calculates the relative displacement of the jump and places this
value in the object code.  Remember that if a number is used, it will
be considered to be an absolute address, NOT a displacement.  Note that
this may be different from the description of relative jumps in the
Z-80 manuals by Mostek and Zilog.  Some examples will illustrate these
concepts; the statement


              JR       NZ,100


tells ASMB to generate a relative jump to LOCATION 100 or 64H, NOT
```

to jump relative to the present location by 100 bytes. To avoid this
confusion a better form would be

```
            JR      NZ,LABEL
            :
            :
    LABEL:  LD      A,3
```

for which ASMB will calculate the correct jump no matter where LABEL
happens to be located. (However, if the label belongs to another type
of Code Segment, an expression error will be generated; for example
it is illegal to jump from a REL area to a DATA area using a relative
jump.)

----
syntax error

This error message covers a wide range of ills; it generally appears
when a quantity in one of the four Assembler Fields (see Chapter 3)
has been misused. For example, writing a remark without preceding
it with a ";" on a line which already contains a label, opcode, and
operand will produce a syntax error. If you don't know the cause of
the message, look up the expression or opcode of which you are unsure.

----
too many COMmons

This message follows the use of more than the allowable number of
COMmons. ASMB allows a total of 15 COMmons including one "blank"
COMmon. The term blank COMmon means that one of the COMmons need
not be named, not that there is nothing in it. See the COM pseudo-op
for more information on their use.

----
undefined symbol

This message follows a line containing a label name in the operand
field which has not been defined. This is one of the most common of
assembly language mistakes: using a label name for a data quantity and
then forgetting to define it. Labels are defined by appearing in the
label field of any opcode or pseudo-op which allows labels.

----
value error

This message follows a line in which a value is used which exceeds the
range allowable for the opcode used. This value may be a constant or
an expression. Opcodes which expect one-byte quantities will generate
a value error for any expression whose value exceeds the range 0 to
FFH (or its equivalent representation in decimal, octal, or binary).
Opcodes which expect two-byte quantities will not generate an error if
the value simply exceeds the numeric range (65,535); the value will
simply "wrap around". That is, a value of modulus 65,536 is returned

without an error flag. Some examples will illustrate these ideas.
The following will generate a value error:

        LD   A,3000H   (a two-byte quantity used as one byte)

However, the line

        LD   HL,70000

will not generate a value error; instead, the value 4464 or 1170H (which
is 70000-65536) will be generated.

Value errors will also be generated by the BIT, SET, and RES opcodes
if the value of the expression used as an operand is outside the
range 0 through 7.

                                                        Page 60

****
CHAPTER 7: ASSEMBLER PRINT-LISTINGS
****

Following is the print-listing which results from the assembly of
the example in Chapter 1 ("Getting Started"). There is much valuable
information in this listing; it is therefore given here in a separate
chapter so that the various terms and symbols can be explained. The
command line which was typed to produce this assembly is slightly
different from the command line typed in Chapter 2. This is because
several Assembler Options have been specified here so the user can
see what type of listing they produce. The command line which was
typed to produce the following assembly is:

        ASMB TIMER SYMB XREF OPCODE RANGE

The SYMB option requests a Symbol Table, XREF and OPCODE request Symbol
and Opcode Cross Reference Tables, and Range requests those absolute
jumps which are within range to be relative jumps. The next four pages
contain the listing that results from this assembly. Following that
is an explanation of the terms and conventions used in the listing.

                                                        Page 61

CROMEMCO CDOS Z8O ASSEMBLER version 02.02   PAGE 0001

                0001 ; This program rings the console bell at approximately
                0002 ;   half-second intervals determined by a timer loop.
                0003 ;
    (0007)   0004 BELL: EQU   7   ; console bell is ASCII 07
    (0002)   0005 WRITE: EQU   2   ; write character to console
    (0005)   0006 CDOS: EOU   5   ; use system call to write

```
        (02FF)          0007 TIMIT:  EQU     2FFH            ; 2 is no. of half-seconds;
                        0008 ;                                  FF (256) is no. of loops
        (00FF)          0009 DURAT:  EQU     0FFH            ; FF (256) is loop duration
                        0010 ;
                        0011 ; Main Program
                        0012 ;
0000' 315A00'          0013 START:  LD      SP,STACK        ; initialize stack pointer
0003' 01FF02           0014 LOOP:   LD      BC,TIMIT        ; B is no. of half-sec.;
                        0015 ;                                  C is no. of loops
0006' 3EFF             0016 TIM2:   LD      A,DURAT         ; get duration (256)
0008' 3D               0017 TIM1:   DEC     A               ; decrement and
0009' 20FD             0018          JR      NZ,TIM1         ; loop til zero
OOOB' 0D               0019          DEC     C               ; decrement loop counter
OOOC' 20F8             0020          JR      NZ,TIM2         ; until zero
OOOE' 10F6             0021          DJNZ    TIM2            ; countdown half-seconds
0010' 1E07             0022          LD      E,BELL          ; set-up to ring bell
0012' 0E02             0023          LD      C,WRITE         ; set-up to write console
0014' CDO500           0024          CALL    CDOS            ; call system
0017, C30300'       R 0025          JP      LOOP            ; loop and repeat
                        0026 ;
                        0027 ; Stack Area
                        0028 ;
001A' (0040)           0029 BOTTOM: DS      40H             ; allow 64 bytes for stack
        (005A')         0030 STACK:  EQU     $               ; current location counter
                        0031 ;                                  equals top of stack
005A' (0000')          0032          END     START


Errors          0
Range Count     1
Parity Count    0

Program Length  005A (90)
```

Page 62

Page 62

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.02                      PAGE 0002
SYMBOL TABLE


BELL    0007    BOTTOM  001A'   CDOS    0005    DURAT   00FF    LOOP    0003'
STACK   005A'   START   0000'   TIM1    0008'   TIM2    0006'   TIMIT   02FF
WRITE   0002
```

Page 63

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.O2                      PAGE 0003
CROSS REFERENCE LISTING


BELL    0004    0022
BOTTOM  0029
CDOS    0006    0024
DURAT   0009    0016
```

```
LOOP     0014  0025
STACK    0030  0013
START    0013  0032
TIM1     0017  0018
TIM2     0016  0020  0021
TIMIT    0007  0014
WRIIE    0005  0023
```

Page 64

CROMEMCO CDOS Z80 ASSEMBLER version 02.02   PAGE 0004
OPCODE CROSS REFERENCE LISTING

```
CALL          0024
DEC           0017 0019
DJNZ          0021
DS            0029
END           0032
EQU           0004 0005 0006 0007 0009 0030
JP            0025
JR            0018 0020
LD            0013 0014 0016 0022 0023
```

Page 65

```
++++++++++++++
Listing Columns
++++++++++++++
```

This listing is divided up into a number of columns or fields. These
are described below.

Column 1 - This is a 16-bit address printed in hex. If an absolute
ORG statement has not been given, the addresses will start with 0.
Since the (nodules are relocatable, however, the Subsequent addresses
are only relative to the final program base when the program has been
loaded. Immediately following the address is either a space or one
of the symbols: ', ", or *. These are described below.

Column 2 - If the statement is a pseudo-op which generates a value
(for example, the EQU statement), that value will be printed here in
parentheses. For all Z-80 opcodes this column will contain up to four
bytes of object code in hex. The DU and DW pseudo-ops will also
produce object code in this column. If the code being assembled is
relocatable, all addresses will correspond to relocatable addresses
in column one, NOT the actual addresses these bytes will have when
the program is linked and loaded into memory. Relocatable addresses
will be followed by one of the symbols: ', ", *, or #, described
below.

Column 3 - This column is usually not printed. If the Range Option
has been specified, all absolute jumps which are within range to be
relative jumps are marked with an "R" character in this column.

Column 4 - This column contains the line numbers of the source code
in decimal beginning with 0001. All lines will be numbered including
those coitaining only remarks.

Column 5 - This is the label field of the original source. See
Chapter 3 of this Part for a complete description.

Column 6 - This is the opcode field of the original source. See
Chapter 3 for a complete description.

Column 7 - This is the operand field of the original source. See
Chapter 3 for a complete description.

Column 8 - This is the remark field of the original source. See
Chapter 3 for a complete description.

+++++++++++++++
Lines of Listing
+++++++++++++++

The listing also contains useful information on the lines printed out
at its beginning and end. These are described below.

Beginning, Line 1 - This line contains the heading of the listing
giving the current version and release numbers of ASMB. Also on this
line is the page number; listings are numbered consecutively in decimal
including the symbol and other tables at the end.

Beginning, Line 2 - This line will contain the title of the module
being assembled if the user specified one using the TITLE pseudo-op.
A blank line is inserted if no title is used. The Assembler also
inserts a blank line just before the listing begins on every page.

Interspersed Lines - Error messages occupy one full line of a listing
and are printed immediately following the line in which the error was
first detected.

End of Listing, Line 1 - The total number of errors which occurred
during Assembly is printed on this line.

End of Listing, Line 2 - This line will be printed only if the Range
Option has been specified, and it gives the total number of jumps
marked by Range.

End of Listing, Line 3 - This line will be printed only if the Parity
Option has been specified, and it gives the total number of 8080-Z80
conflicts found (see Parity Option in Chapter 2).

End of Listing, Line 4 - This gives total program length (ie, the
byte-count of the object code) in both hex and in decimal.

End of Listing, Line 5 - This and the following lines list all those
COMmons which have been defined in the module along with their lengths

in both hex and in decimal. Up to 15 COMmons may be listed.

```
++++++++++++++
Listing Symbols
++++++++++++++
```

There are four symbols which appear throughout a print-listing which
give some additional information. These are described below.

Single Quote (') - This symbol follows all addresses (column 1) which
belong to a REL area. The symbol also follows all references to REL
addresses made in the object code. For example the three bytes:

        C31F00'

in the object code mean to jump to address 001F of the REL segment of
code.

Double Quote (") - This symbol follows all addresses (column 1) which
belong to a DATA area. The symbol also follows all references to DATA
addresses made in the object code. Remember that DATA program segments
are very similar to REL program segments.

Asterisk (*) - This symbol follows all addresses (in column 1) which
belong to a COMmon program segment. The symbol also follows all
references to COMmon addresses made in the object code.

                                                    Page 67

Pound Sign (#) - This symbol appears only following an address in the
object code, and marks those lines as ones referencing EXTernals. The
address just preceding the pound sign is the location that that EXT
was last referenced, or is 0000 if it's the first time in the module
that the EXT is referenced.

Note that addresses in column 1 or in the object which are not followed
by one of the four symbols above belong to an ABSolute segment of code.

```
+++++++++++++++++++++++++++++
Tables Following the Listing
+++++++++++++++++++++++++++++
```

There are several tables which may follow the print-listing of the
source code. These are described briefly below.

```
------------
Symbol Table
------------
```

The symbol table contains an alphabetical list of all the symbols
(labels) defined in the source program. Each symbol will be followed
by its value and one of the four symbols described above to tell the
user to which program segment it belongs. The value will be either
the address at which the symbol is defined or the value of the
expression to which it equates. Note that EXTernals listed in the
symbol table do not follow this rule. The address listed following

an EXT name is the address of its first occurrence in the source.

Cross Reference Table

The cross reference table contains an alphabetical list of all symbols
and the line numbers of both their places of definition and occurrence
throughout the source program. The symbols are listed in the first
column, the line numbers of their definition in the second column, and
the line numbers of their occurrence are listed by rows to the right
of the first two columns. Symbols which have been multiply-defined
by the use of DL statements will have the line numbers of subsequent
definitions listed to the right and followed by the pound sign (#).

Opcode Cross Reference Table

The opcode cross reference table contains an alphabetical list of all
opcodes and Macro names along with the line numbers of their places
of occurrence (and places of definition for for Macros). The opcodes or
Macro names are listed in the first column, the line numbers of Macro
definitions ONLY are listed in the second column, and the line numbers
of their places of occurrence are listed by rows to the right.

This completes the description of the items which make up an assembled
print-listing. This also completes Part I of this book.

                    ****
                    PART II - CROMEMCO LINKER/LOADER MANUAL
                    ****

```
*********************************************
CHAPTER 1:  USING THE CROMEMCO LINKER/LOADER
*********************************************
```

```
++++++++++++++
Command Format
++++++++++++++
```

The CROMEMCO Linker/Loader is used to link assembled program modules
together, load them into memory, and begin execution there if desired.
The Linker is supplied to the user on diskette (large or small) under
the directory entry "LINK.COM". The command line to call LINK consists
of a number of filenames and switches according to the following format:

```
        LINK <d:filenam1.ext/s,d:filenam2.ext/s,...)
```

where d stands for the disk drive letter (A through D), s stands for
one of the legal switches of the Linker (see list in this chapter),
and filename.ext stands for a user filename plus its 3-letter
extension. The only quantity required above after the word LINK
is filenam1. LINK defaults to the current drive if the disk drive
letter is omitted, and it defaults to the extension .REL if the 3-
letter extension is omitted. The switches are not necessarily
required, and are used to give LINK instructions regarding the files.
The Linker will accept commands in the order received, but does not
require a single command line. The prompt for LINK is an asterisk,
"*", any time the asterisk appears, a command may be entered. Thus,
the names of files to be linked may be given one at a time rather
than an one command line. The example of Chapter 4 will illustrate
this further. After each line is typed, LINK will load or search
the named file(s). When LINK finishes this process, it will list
all symbols that remain undefined followed by an asterisk.

The switches LINK accepts give the user a variety of ways to control
the linking process. For example the user may cause the Linker to
search special library files to satisfy undefined globals by linking
the filename to be searched followed by /S. The /M switch can be used
to map a list of all defined and undefined symbols. These switches
are described in the next section. Chapter 2 gives a brief explanation
of the operation and format of LINK and associated .REL files for
those who are interested. It may be safely skipped, however, for it
contains no information on the actual use of the Linker. Chapter 3
is a brief summary of the error messages that occur and why, and
Chapter 4 gives a step-by-step example of the process of linking and
loading program modules.

```
+++++++++++++
LINK Switches
+++++++++++++
```

The Linker allows a number of switches which specify actions affecting
the loading process. These switches are listed here.

```
-----------------
/E (Exit to CDOS)
-----------------
Exit to CDOS upon completion of link and load.  Prior to exiting, LINK
prints on the console the start and stop execution addresses along
with tile number of 256-byte pages of memory the program occupies (in
decimal), according to the following format:
```

        [xxxx yyyy zz]

where xxxx is the address at which execution will start, yyyy is one
more than the highest location used by the loaded object code, and zz
is the decimal number of pages required.

If it is desired after executing the /E to save the file now located
in memory, this can be done using the SAVE command, which is one of
the CDOS intrinsic commands (see also CDOS manual). The user would
then type:

        SAVE filename.ext zz

where zz is the same number printed out by LINK above (following the
issue of /E). The filename can be any legal name; however, if the name
used already resides on the disk, the saved file will be written over
this existing file. The 3-letter extension is frequently .COM because
this procedure is often used to create command files; however, any
extension may be given. Note that other CDOS INTRINSIC commands may be
given before the SAVE command; for example, DIR may be typed to see
about available directory space. However, executing any EXTRINSIC
commands (XFER, EDIT, etc.) will change the contents of the user-area.
For a 32K system, zz=105 will save the entire user-area.

```
-------------------------
/G (Go - start execution)
-------------------------
```
Start execution of the program as soon as the current command line has
been interpreted. Prior to execution, LINK prints on the console the
start and stop addresses and the number of 256-byte pages occupied by
the object code, according to the format shown above (see /E). Fol-
lowing this is the message "[BEGIN EXECUTION]" at which point execution
is started by LINK. The Linker initializes the stack pointer at the
highest address of the user-area in case this operation is forgotten
by the user program.

```
--------------------
/M (Map all symbols)
--------------------
```
List both all the defined globals and their values and all undefined
glabals followed by an asterisk. The map may be sent to the printer
by typing Control-P (^P) following the LINK command line. This printer
map of symbols is very useful for debugging the user-program. Once
the object code has been loaded into memory by LINK, /E can be issued
and the correct portion of the user-area saved in a file. Then the

program DEBUG can be called and used to load and debug the file just
created. The global map printed previously can be used to reference
addresses.

/R (Reset linker)

Put Loader back in its initial state. /R is used to restart LINK if
the wrong file was loaded by mistake. /R will take effect as soon as
it is encountered in a command string.

/S (Search file)

Search the disk file having the filename immediately preceding the /S
in the command string, to satisfy any undefined globals. This is
convenient for having the Linker search a library file of much-used
routines. (Note that when using LINK with CROMEMCO FORTRAN, the
library file FORLIB.REL is searched automatically to satisfy undefined
globals.)

/U (list all Undefined globals)

List all undefined globals as soon as the current command line has been
interpreted and executed. LINK defaults to this switch; therefore, it
is generally not needed unless it is desired to reproduce this list
more than once. For example say that during link the list of undefined
globals is printed to the console. The user could then type Control-P
followed by "/U" to cause the undefined globals to be listed a second
time, this time to the printer as well as the console.

****
CHAPTER 2: FORMAT OF LINK-COMPATIBLE OBJECT FILES
****

The following is a description of the format of REL files which
are to be compatible with the CROMEMCO Linker. This information
is provided for the interested programmer, but is not in any way
required reading for the person learning how to USE the Linker.

LINK compatible object files consist of a bit stream. Individual
fields within the bit stream are not aligned on byte boundaries
except as noted below. The use of a bit stream for relocatable
object files keeps the size of the files to a minimum, thereby
decreasing the number of disk reads and writes. The first bit
of a field is either a one or a zero, and this is followed either
by an 8-bit byte or a 2-bit field having the following meanings:

Bit   Meaning
0   (load the following eight-bit byte as absolute code)
1   (read in the following two bit field:)

```
              11  Add sixteen bit offset to common base
              10  Add sixteen-bit offset to data base
              01  Add sixteen-bit offset to program base
              00  Special LINK item
```

Special LINK item fields begin with the bit stream 100 as just
explained. This is followed by a four-bit control field, an
optional A-field which consists of a two-bit code specifying address
type, and an optional B-field which consists of 3 bits giving a
symbol length. The 2-bit address type has the same meanings as
the 2-bit field above except 00 specifies absolute addressing.
The 3-bit symbol length is followed by eight bits for each character
of the symbol. We can represent this bit stream by the following:

```
                A-field                 B-field
1 00 xxxx <yy two-byte-value> <zzz characters-of-symbol--name>
```

where the spaces in the above show where the various fields end, the
angular brackets denote optional quantities, and where

```
xxxx  is the four-bit control field
yy    is the two-bit address type field
zzz   is the three-bit symbol length field
```

The two-byte-value following yy will be either the 16-bit offset
specified or the absolute address, and the characters-of-symbol-name
following zzz will be in ASCII, each character occupying eight bits.

                                                 Page 73

The four-bit control field will specify the operation or function
of the bit stream. It can have the following values, where the
four-bit value is given in the left-hand column in decimal:

(The following LINK items have a B-field only:)

```
0        Entry Symbol (name for search).
1        Select COMmon Block.
2        Program Name.
3        Reserved for Future Expansion.
4        Reserved for Future Expansion.
```

(The following LINK items have both an A-field and a B-field:)

```
5        Define COMmon Size.
6        Chain External (A is head of address chain).
         B is name of external symbol.
7        Define Entry Point.
8        Reserved for Future Expansion.
9        Reserved for Future Expansion.
```

(The following LINK items have an A-field only:)

```
10       Define Size of Program Data Area.
11       Set Londing Location.
12       Chain Address.
```

A is head of chain; replace all entries in chain with
current location counter. The last entry in the chain
has an aldress field of absolute zero.
13   Define Program Size.
14   End Program (forces to byte boundary).

(The following LINK item has neither an A- nor a B-field:)

15   End of File.

```
******************************
CHAPTER 3:   LINK ERROR MESSAGES
******************************
```

The Linker gives several error messages in case of an illegal
operation. These are listed below in the summary along with an
explanation of each one. Note that there are two types of error
messages: fatal errors and warnings. Fatal error messages are
preceded by question marks (?) and warning messages are preceded
by percent signs (%). A program will run in some cases when a
warning has been issued; however, it is better practice to correct
the error and link again.

Fatal Errors

?No Start Address	A /G switch is issued, but no main program module has been loaded. Remember when creating and linking machine language programs that the main module must have an address or label in its END statement. This then becomes part of the .REL file which informs LINK where to begin execution (see also the END pseudd-op).
?Loading Error	The last file given to be linked and loaded is not a properly for-matted LINK object file.
?Fatal Table Collision	There is not enough memory to load the given program(s)
?Command Error	An unrecognizable LINK command has been given. Type the correct command or re-link.
?File Not Found	A file in the command string does not exist as spelled or specified. Check to see if the file resides on the specified drive. Often this message results if the user forgets to specify the drive letter, and LINK looks on the current drive.

Warnings

%2nd COMMON Larger /XXXXXX/   The first definition of COMmon
                              block XXXXXX is not the largest.
                              COMmons do not have to be the
                              same size provided the module
                              containing the larger COMmon
                              specification is linked first so
                              that LINK allocates an appropriate
                              number of bytes for data storage.
                              To prevent this error re-order the
                              module loading sequence or change
                              the COMmon block definitions.
%Mult. Def. Global YYYYYY   More than one definition for the
                              global (internal) symbol YYYYYY
                              is encountered during the loading
                              process. This message may result
                              if you redefine the LUN table of
                              FORTRAN ($LUNTB) and then link
                              with FORLIB.REL without specifying
                              the /S switch. The Linker then
                              loads both the redefined version
                              of $LUNTB and the version contained
                              in FORLIB.

****
CHAPTER 4: EXAMPLES OF LINKING MODULES
****

Following are several examples of the process of linking, loading,
saving, and executing files. The asterisk (*) in the following
command lines is NOT user-typed; it is the prompt for LINK.

We would type the following command to load a 32-byte program called
MYPROG into memory and begin execution:

        LINK MYPROG/G

If the load is successful (no errors), the Linker will respond with
the message:

        [1000   1020   16]
        [BEGIN EXECUTION]

This program will begin execution at 1000H. If we desired to save the
program prior to execution, could type instead:

```
      LINK MYPROG/E
```

to which the Linker would respond with:

```
      [1000    1020    16]
```

followed by a return to CDOS and the issue of the CDOS prompt. This
return to CDOS does not change the user area; hence, we could then
save the program by typing:

```
      SAVE MYPROG.COM 16
```

Since we have named this a .COM file, we can execute it directly
from CDOS by typing the nane "MYPROG".

Another example would be to link several modules together as they
are loaded into memory. Suppose we have the three relocatable
modules GRAPHX, MAIN, and SUBPLOT. We first type:

```
      LINK <CR>
```

to which LINK responds with the asterisk. We could then type:

```
      MAIN
```

The Linker would look on the current drive for MAIN and then return

the still-undefined symbols (each one followed by an asterisk) and
the address at which they are referenced:

```
      INITG*  122E
      LINE*   164D
      CURSR*  163E
      STRIN*  131B
      SUBROT* 147D
      *
```

We then link the next module:

```
      GRAPHX
```

and LINK again responds with the undefined symbols and the prompt:

```
      SUBROT* 147D
      *
```

Finally, we link the last module:

```
      SUBROT
```

to which link responds with the prompt. We could now type /G or /E
to run or exit from the program as we did in the first example.
However, let's first generate a map of all the symbols using the /M

LINK switch:

        */M

to which the Linker would respond:

        INITG* 122E
        LINE*   164D
        CURSR* 163E
        STRIN* 131B
        SUBROT* 147D
        PAGE   17DF
        DOT   180E
        ANIMT   1558

Note that this is similar to the map of undefined symbols; however, in this case symbols which are not used, but have been defined in one of the linked programs, are also listed.

The above example could also have been linked directly, and without producing the maps of undefined symbols, by typing the command line:

        LINK GRAPHX,SUBPLOT,MAIN/M

Note also that this command line links them in a different order than the first case since all of the modules are relocatable. Thus, the map printed to the console this time would have a different address after each symbol.

The Linker can also be used to link machine language subroutines to programs written for and compiled with CROMEMCO FORTRAN IV. The assembly language subroutine should be assembled with ASMB, which forms a .REL file. The form of the link is then exactly the same as for the previous example. An important note is that LINK has been designed to automatically search FORLIB.REL, the FORTRAN Library file of subroutines. LINK looks for this file on drive A, rather than the current drive. The user can force the Linker to look for FORLIB on another drive by typing a command like:

        LINK FORTRAN,SUBROT,B:FORLIB/S

where FORTRAN is the user's compiled FORTRAN program. Note the use of the /S switch following FORLIB. This tells LINK to load into memory only those routines which are actually needed rather than the entire Library. It is important to use this switch with library files in order to save memory space.

Finally, note that the user may return to CDOS at any time while using LINK (to abort the linking or loading process, for example) by typing Control-C (^C).

```
*******************************************
PART III - CROMEMCO PROGRAM DEBUGGER MANUAL
*******************************************
```

Page 80

```
******************************
CHAPTER 1: INTRODUCTION TO DEBUG
******************************
```


The CROMENCO DEBUG program makes it possible to test and debug
user programs. DEBUG is loaded into memory and moved to the highest
memory available below CDOS. When using a 32K CDOS and DEBUG, there
is 20K left for the user program.


```
+++++++++++++
LOADING DEBUG
+++++++++++++
```


DEBUG is loaded by typing one of the following commands from CDOS.

```
        DEBUG
        DEBUG filename.ext
```

where "filename" is the name of the program to be tested, and "ext" is
the file extension. In both cases, DEBUG is loaded into memory directly
below CDOS. The CDOS jump instruction located at location 5H is changed
to jump to the start of DEBUG. This allows locations 6H and 7H to still
point to the lowest available memory location.
        The second command above is used to load the file to be tested into
memory. If the extension ("ext") is ".HEX", then the file is read as an
INTEL HEX file. Any other extension is read as an absolute binary file,
loaded at location 100H. **** NOTE **** DEBUG does not load relocatable

files. If an extension is ".REL" it will be loaded in as if it were
binary and will not be executable.

+++++++++++++++++++
CONTROL CHARACTERS
+++++++++++++++++++

     Control characters are used in DEBUG and TRACE to help in entering
commands. These control characters are the same as CDOS uses.

          Control-C (^C)   go back to CDOS
          Control-H (^H)   delete character and backspace on CRT
          Control-U (^U)   delete line
          Control-X (^X)   delete character and echo
          underscore   delete character and backspace on CRT
          RUBout (DEL)   delete character and backspace on CRT

     During a printing (such as from the DM command) the following
characters may be used.

          Control-S (^S)   stop/start printing. If printing, this
                             character will stop the printing. If already
                             stopped, this character will resume the
                             printing.

          break   (or any other character) will abort the
                             printing, prompt, and wait for the next
                             command.

++++++++++++++
COMMAND FORMAT
++++++++++++++

     DEBUG is controlled by one and two character commands from
the terminal. The format is free-form in respect to spaces. Commas
may be used in place of spaces. In the following, the examples all
dump memory starting at location 1000H and ending at location 10FFH.

               DM1000 10FF (CR)
               DM1000S100 (CR)
               D M 1000   10FF (CR)
               D M 1000 S 100 (CR)
               DM1000,10FF (CR)
               DM1000,S100 (CR)
               D M 1000 , 10FF (CR)

++++++++++
@ REGISTER
++++++++++

     DEBUG was designed to give flexibility in testing relocatable
programs. The "@" register is used to tell DEBUG where the module

you wish to debug is located. This address can be found from
the map generated by the linking loader "LINK". To change
the "@" register, type "@ (CR)" on the console. The computer will then
type "@-xxxx ", where xxxx is the current value of the register. The
computer will then wait for a new address. If a CR only is typed, the
register remains unchanged. If an address and a CR is typed, then the
register will contain the new address. The "@" register may now be
used as part of an address. The following example demonstrates it's
use.

                    G/@ @A3 1000

        This is an example of the go command. Break points will be set at
the beginning of the current module, relative location A#H in the
current module, and at location 1000H. This feature allows you to test
a module without having to calculate absolute addresses.

++++++++++++++++++
ADDRESS EXPRESSIONS
++++++++++++++++++

        For additional ease in specifying addresses an expression can be
used. Within these expressions, addition, subtraction, the "@" register,
and the "$" may be used. The "$" is the current location of the program
counter (P register). If many modules are being tested, addition can be
used to specify relative addresses.

                    G/2321+A3

        The preceeding example would set a break point at relative location
A3H if the module is located at 2321H.

+++++++++++++
SWATH OPERATER
+++++++++++++

        There are two ways to specify the address range of many commands.
The first is to simply list the beginning and end addresses (and where
appropriate, the destination address). For example, the first command
below programs the range 0 through 13FFH into PROMs starting at location
E400H. The second command displays the contents of memory between
addresses E400H and E402H.

                    PO 13FF E400
                    DME400 E402

        Another way to do the same thing is to use the Swath operator, "S",
to specify the width of the address range, rather than state the end
address explicitly.

                    PO S14OO E400
                    DM E40OS3

++++++

ERR0RS
++++++

       Any errors made during entering of a command may be corrected by
typing Control-U (^U) to abort the line or by backspacing and correcting
the line. If a CR has already been entered and DEBUG detects an
error, the line will not be accepted and a "?" will be printed.
Re-enter the line with the incorrect data corrected.

****
CHAPTER 2:   DEBUG COMMANDS
****

       DEBUG and TRACE commands are described in detail below.   The
operator must wait for the prompt character ("-") before entering the
command.

A - Assemble into memory

       This command allows in-line assembly language to be assembled into
memory.   The command takes the following format.

                  A beginning-addr (CR)

       The user is prompted with the absolute address, followed by the
relative address.   DEBUG reads from the console the assembler mnemonics,
and assembles the instruction into memory.   The mnemonics for the various
Z-80 instructions can be found in the Z-80 CPU TECIINICAL MANUAL published
by Mostek and Zilog. If there was no error in the instruction it is
stored in memory and the User is prompted for the next instruction.   The
The rules for address expressions apply to the addresses in the assembler
mnemonics.   In the following example the "@" register contains 1234H.

                  A@40
                  1274 0040'   ADD B
                  1275 0041'   CALL @93
                  1278 0044'   JP 1032+95
                  127B 0047'   .

       The A command terminates when the first blank line or a line
starting with a "." is entered from the console. If there is an error
in the input line, it will not be accepted, a "?" will be printed and
the console will be prompted with the addresses again.

DM - DISPLAY MEMORY

       The contents of memory are displayed in hexadecimal form. Each
line of the display is preceded by the address of the first byte and
followed by the ASCII representation of the hexadecimal bytes. An
example follows

```
DM100,S30
0100   40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
0110   50 51 52 53 54 55 56 57-58 59 5A 30 31 32 33 34   PQRSTUVWXYZ01234
0120   35 36 37 38 39 00 00 00-00 00 00 00 00 00 00 00   56789...........
```

     The formats of this command are as follows.

               DM (CR)
               DM beginning-addr (CR)
               DM beginning-addr ending-addr (CR)
               DM beginning-addr S swath-width (CR)
               DM,ending-addr (CR)
               DM S swath-width (CR)

     The first format displays memory from the CURRENT display address,
initially 100H, and continues for 8 lines. The second format displays
from the beginning address and continues for 8 lines. The third format
displays from the beginning address to the ending address. The forth
format displays from the beginning address for a length specified by
the swath-width. The fifth format displays from the CURRENT display
address to the ending address. The sixth format displays from the
CURRENT display address for a length specified by the swath-width.
     If an "X" is included after the "DM", the relative addresses are
also printed. In the following example assume that the "@" register
contains 100H.

```
DMX100,S30
0100 0000' 40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
0110 0010' 50 51 52 53 54 55 56 57-58 59 5A 30 31 32 33 34   PQRSTUVWXYZ01234
0120 0020' 35 36 37 38 39 00 00 00-00 00 00 00 00 00 00 00   56789...........
```

DR - DISPLAY REGISTERS

     When DEBUG or is re-entered from a break point, the user registers
are saved. The registers may be displayed at any time by typing the
following command.

```
-DR (CR)
SZHVNCE A=00 BC=0000 DE=0000 HL=0000 S=0100 P=0100' LD   E,A
SZHVNC A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

     The letters "SZHVNC" are the flags, on the second row are the prime
flags. If the flag is on, it is printed, if the flag, is off, a space is
printed. If only the carry and zero flag are set then " Z   C" would be
printed. The flags are described below.

          S - Sign flag, S=l if the MSB of the result is one, ie, the
               result is negative.
          Z - Zero flag, Z=l if the result of an operation is zero.
          H - Half-carry flag, H=1 if the add operation produced a carry
               into the 4th bit of the accumulator or a subtract operation
               produced a borrow from the 4th bit of the accumulator.

V - Parity or overflow flag. This flag is affected by arithmetic
             and logical operations. If an overflow occurs during an
             arithmetic operation, the flag is set to one. After a
             logical operation, the flag is set 1 if the result of
             the operation has even parity.

         N - Add/subtract flag, N=1 if the last operation was a
             subtraction.
         C - Carry flag, C=1 if the operation produced a carry.

     The E flag on the first line is the state of the interrupt enable
flip-flop (IFF). If interrupts are enabled, the "E" is printed,
otherwise a space is printed.
     The A register is printed next, followed by the BC, DE, and HL
register pairs and the stack pointer. The program counter value is then
printed in both absolute and relative. The opcode pointed to by the
program counter is then displayed as an instruction.
     On the second line, the prime registers are displayed, F' (prime
flags), A', BC', DE', and HL'. The IX, IY, and I (interrupt page)
registers are printed next. If the disassembled opcode includes an
address, the relative value of this address is printed as the last thing
on the line.

-DR (CR)
S H NCE A=00 BC=0000 DE=0000 HL=0000 S=0000 P=1234 0010' CALL 1334
SZ NC A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00   (0110')

E - EXAMINE INPUT PORT

     The data port is read and displayed as a hexadecimal number. The
format of the command is

        E data-port (CR)

     In the following example the data part 3 is read and displayed on
the console.

        -E3 (CR)
        23

EJ - EJECT DISK

     The format of the command follows.

        EJ d

     The d is the disk number (A, B, C, D). If the designated disk is
a CROMEMCO DUAL DISK SYSTEM model PFD, with the eject option, the
diskette in the disk drive will eject.

```
--------------------
F - SPECIFY FILE NAME
--------------------


     This command allows the operator to insert filenames in the two
default FCBs (at 5CH and 6CH) and the command line into the default
buffer (at 80H).  The example below loads FILE1.COM into the first FCB
and FILE2.COM into the second FCB.  The complete line is also loaded
into the default buffer.

          -FFILE1.COM FILE2.COM OPTION1 OPTION2

     This command can be used with the "R" command to read in disk files.


------
G - GO
------


     The GO command has the following format.

          G(starting-addr)/(breakpoint-1) (breakpoint-2)...(breakpoint-5)

     Each of the addresses are optional.  If the starting address is
omitted, then the contents of the Program Counter is used.  The registers
are loaded from the user registers (these are the values displayed with
the DR command).  Execution begins with the starting address or the
contents of the program counter.  If break points were specified, a
RST 30H is inserted at the break point addresses and a jump instruction
is placed at location 30H.  When a breakpoint is executed, control is
returned to DEBUG, and all of the user registers are saved (the registers
may then be displayed with the DR command).  All breakpoints are then
removed from the user program.  The program counter is displayed after
the breakpoint.  Note the following about breakpoints:

          (a) Breakpoints can only be set in programs residing in RAM.
This is because a RST 30H is inserted at each break point location.
(The original contents of these locations are saved so that they can
be restored after a break point is executed.)
          (b) Up to 5 break points can be set.  If an attempt is made to
enter more than 5 break points, the command will not be accepted.
          (c) When a break point is used, a jump instruction is stored at
location 30H.  Therefore locations 30H, 31H, and 32H are not available
to a user program.

     The GO command has an additional feature that is very helpful in
debugging a program.  A count is allowed for each break-point.  This
count is entered after the break-point and enclosed in parentheses.
This count is the number of times the program reaches this address
before control is returned to DEBUG.  A count of one says to break the
next time the address is reached.  In the example below execution begins
at location 100H and will break when address 109H is reached for the
second time or when 123H is reached for the first time.

     -G100/109(2) 123
```

Note that 123 and 123(1) means the same thing. Also note that the
count is a hexidecimal number. Therefore 123(F) means to break after the
address has been executed for the 15th time.

     H - HEXADECIMAL ARITHMETIC

     Hexadecimal addition and subtraction may be performed by this
command. The first number to be printed is the sum of the two input
numbers. The second number to be printed is the difference between the
first number and the second number. In the example following, the first
number is 1234 + 321, and the second number is 1234 - 321.

          -H1234,321
          1555 0F13

     L - LIST IN ASSEMBLER MNEMONICS

     The list command is used to list the contents of memory in
assembly language mnemonics. The formats for this command are.

          L (CR)
          L starting-addr (CR)
          L starting-addr ending-addr (CR)
          L starting-addr S swath-width (CR)
          L,ending-addr (CR)
          L S swath-width (CR)

     The first format lists 16 lines of disassembled code starting from
the current list address. The second forat lists 16 lines from the
starting address. The third format lists from the starting address to
the ending address. The fourth format lists from the starting address
for a length specified by the swath width. The fifth format lists from
the current list address to the ending address. The sixth format lists
from the current address for a length specified by the swath address.
     The first address of the disassembly is the absolute address. The
second address is the relative address. If the disassembled instruction
contains an address, the absolute address is printed in the instruction
in hexadecimal and the relative address is printed to the right of the
disassembled line. In the example that follows, the "@" register
contains 2800H.

          -L@800 812
          3000 0800' ADD B
          3001 0801' CALL 3200   (0A00')
          3004 0804' CALL 3243   (0A43')
          3007 0807' CALL 3333   (0B33')
          300A 080A' LD   A,B
          300D 080B' OR   C
          300C 080C' JR   Z,3000   (0B00')
          300F 080F' INC HL
          3010 0810' INC DE
          3011 0811' INC BC

```
      3012 0812'  LD   A,H
```

M - MOVE MEMORY

     The formats of this command follow.

        M source-addr source-end destination-addr
        M source-addr S swath-width destination-addr

     The first format moves the contents of memory beginning with the
source address and ending with the source-end to the destination
address. The second format uses the swath width to determine the length
of the move.
     The move is verified to insure that all bytes were moved correctly.
If an overlapping move was made, errors will be reported. The error
reporting can be terminated by typing any character.
     The move command can be used to fill a block of memory with a
constant. In the following example, a zero has been entered into
location 100H using the SM command. The following command will move
zeros from location 100H through 108H.

        -M100 S7 101

     Care should be taken not to move memory over DEBUG, TRACE or CDOS.

----
0 - OUTPUT TO DATA PORT

     This command outputs data to a data port. The following is the
command format.

        O data-byte port-number (CR)

P - PROGRAM PROMS

     This command allows programming of PROMS. The following are the
command formats.

        P source-addr source-end destination-addr
        P source-addr S swath-width destination-addr

     The first format programs PROMs starting with the source address
and ending with the source-end into PROMs beginning at the destination
address. The second format determines the length from the swath width.
     If the length of the source is not a multiple of 400H or if the
destination does not begin at a 400H boundry DEBUG will reject the
command. (Multiples of 400H end in '000', '400', '800', and 'C00'.)
     Any number of 2708 or 2704 PROMs can be programmed in the execution
of one command as long as there are enough BYTESAVERS to contain them.
Each PROM is verified with its source after all are programmed and any

discrepancies are printed out. If no discrepancies are found, a prompt
is printed and the next command may be entered.

     Software can be loaded into a PROM in as small increments as you
desire, provided it is added to previously unused areas of the PROM.
This is done by first using the Move command, "M", to transfer the
contents of the PROM to RAM, adding the new software to an area of RAM
which corresponds to the unused portion of the PROM and finally using
the Program command, "P", to reprogram the PROM with the result.
Although the entire PROM must always be programmed, it never hurts to
rewrite the same data over again. In general, a 1 may be written over a
1, a 0 over either a 1 or a 0, but the only way to change 0's to 1's is
to erase the PROM with appropriate UV light. (See the BYTESAVER manual
for details.)

R - READ DISK FILE

     This command allows the operator to read a disk file. The "R"
command is used with the "F" command. The "F" command is used to
specify the filename, and the "R" command reads in the file. If the
file has an extension of ".HEX", then the file is an INTEL HEX file and
will be read into memory. Any other file is considered to be a binary
file and will be read directly into memory beginning at location 100H.
The format of the "R" commands is

     R
     R displacement

     The first format reads the file with no displacement. The second
format reads the file with a displacement. If the input file is in HEX,
then the displacement is added to the addresses in the file to
determine the addresses at which to store the file. If the file is a
binary file, it will be stored at the displacement + 100H.
     When the "R" command is executed, DEBUG prints either a "?" if
there is an error (file not found, checksum error. or file attempting
to read above highest available memory location) or with the following
message if there is no error.

     NEXT = xxxx

     Where xxxx is the address of the next available memory location
past the end of the file.

SM - SUBSTITUTE MEMORY

     This command is used to substitute memory. The format of the
command follows.

     SM starting-addr

DEBUG prints the absolute address, followed by the relative address, followed by the contents of the memory byte. One of the following may then be entered.

(a) data-byte value. The data byte value is stored at the
    address of the prompt. The address is then incremented
    by 1 and displayed on the next line.
(b) string enclosed in quotes. The string is stored beginning
    at the address of the prompt. The address is then incremented
    past the string and displayed on the next line.
(c) Any number of (a) and (b) above can be entered on one
    line. The address is then incremented past the bytes that
    were stored and the now address is displayed on the next
    line.
(d) "-". A minus sign does not store a byte. The address
    will be decremented to the previous address. The minus
    sign can be used to "back up" to a previous location in
    case an error has been made.
(e) (CR) only. If no entry is made on the line, the memory
    byte remains unchanged. The address is incremented by 1 and
    displayed on the next line.
(f) period. A period ends the input mode and returns to the
    command level.

In the example that follows, assume that the "@" register contains the value 2800H.

```
-SM@100
2900 0100' 32 0
290l 0101' 17 00
2902 0102' 31 'THIS IS AN ASCII STRING'
2919 0119' 7A 'AAAA' 0 0 1 2 3 4 5 6 7 8 9
2928 0128' 22
2929 0129' 29
292A 012A' 87 -
2929 0129' 29 .
```

SR - SUBSTITUTE REGISTER

The Sr command allows the user registers to be altered. The letter "r" stands for the register which is to be changed. The section SUMMARY OF REGISTER NAMES gives a summary of the names that can be substituted. When substituting the F and F' flags, enter the command SF or SF'. DEBUG will then print the flags that are set and wait for the operator to enter the names of the registers that are to be set. If the flags are not entered, the flags are reset. In the following example, the "SZHC" flags are set. After the example is executed the "ZC" flags are set. The lower case, letters are entered by the operator.

```
-sf
SZH  C  zc
```

When sustituting a one byte register, a one byte value is accepted.
When substituting a two byte register, a two byte value is accepted. If
no value is entered, or if an error occurs, the value of the register
remains unchanged. In the following example, the A register is changed
to contain 41H.

        -sa
        A=98 41

                                                            Page 91

T - TRACE

     The format of trace is

        T (CR)
        T number-of-lines (CR)

     The first format traces the program through one instruction. The
second format traces the program through "number-of-lines"
instructions. After every instruction traced, the values of the user
registers are printed in the same format as the "DR" command.
     You can trace only through RAM. The trace command places a break
point after the instruction, loads the registers and executes the
instruction. The break point is then executed and the registers are
resaved. The registers are printed, and the next instruction is
executed unless the count has reached zero, in which case a prompt is
printed and you may enter the next command.
     To abort the trace, hit any key on the console. A prompt will be
printed and you may enter the next command.

TN - TRACE WITH NO PRINTING

     The "TN" command is the same as the "T" command with the exception
that after every instruction is traced, the registers are not printed.
Only the last traced instruction is printed.

V - VERIFY MEMORY

     Verify that the block of memory between source address and source
end contain the same value as the block beginning at destination
address. The addresses and contents are printed for each discrepancy
found. The following is the format of this command.

        V source-addr source-end destination-addr
        V source-addr S swath-width destination-addr

     This command works by reading bytes from the source and destination
and comparing them. If a discrepancy is found, the memory is read again
for print-out. Thus, it can happen that a discrepancy is printed-out
with the source and the destination contents indicated to be the same.

This is caused by a defective memory element.
     A discrepancy is printed in the following order, source address,
source contents, destination contents, destination address. In the
example that follows, memory locations 1003H and 1008H are defective.

```
        -V 0 S30 1000
        0003 32 12 1003
        0000 7A 5A 1008
        -
```

```
**********************************
CHAPTER 3: SUMMARY OF DEBUG COMMANDS
**********************************
```


     The following is an alphabetical list of the DEBUG commands.

Command	Description
A	Assemble into memory
DM	Display Memory
DR	Display Register
E	Examine input port
EJ	EJect disk
F	specify disk File name
G	Go
H	Hexadecimal arithmetic
L	List in assembler mnemonics
M	Move memory
0	Output to data port
P	Program PROMs
R	Read disk file
SM	Substitute Memory
Sr	Substitute register
T	Trace
TN	Trace with No print (DEBUG only)

V   Verify memory

```
++++++++++++++++++++++++++
SUMMARY OF REGISTER NAMES
++++++++++++++++++++++++++
```

The following register names are printed by the DM command and should
be used with the Sr command.

Register	Description
--------	-----------

F   Flags, the following flags may be changed.
                   S - Sign flag
                   Z - Zero flag
                   H - Half carry flag
                   V - parity/oVerflow flag
                   N - subtractioN flag
                   C - Carry flag

             The interrupt enable flag ("E") may also be changed.

F'   The F' flags are the same as the "F" flags.
             (note that the "E" flag may not be changed here.)

A   accumulator

A'   prime accumulator

B   BC register pair

B'   BC' register pair

D   DE register pair

D'   DE' register pair

H   HL register pair

H'   HL' register pair

S   Stack pointer

P   Program counter

X   IX register

Y   IY register

I   Interrupt page register

```
*********************************************
CHAPTER 1:  INTRODUCTION TO CDOS SYSTEM CALLS
*********************************************
```

This section of the manual descibes the use of CDOS system
calls. CDOS handles disk files, performs device input and
output, and contains a number of useful subroutines.

```
++++++++++++++++
Memory Allocation
++++++++++++++++
```

CDOS resides in high memory. It reserves memory below 100H for
its own use. The user is left all memory from 100H to the
beginning of CDOS (see below).

A program with the extent ".COM" can be loaded and executed by
merely typing the program name. The program must have its origin
at 100H because that is where CDOS loads and executes it. (Note
that when saving files that have been linked using the CROMEMCO
Linker, they can be ORGed anywhere because LINK automatically puts
the correct jump instruction at 100H.) After it is loaded, the
program can use any memory at all. Note however, that if it alters
the CDOS areas, it will have no way of communicating with the
disk or returning to CDOS. (CDOS would have to be reloaded by
resetting the computer.)

CDOS places a jump instruction at bytes 0, 1 and 2. If a jump
is made to location 0, the CDOS warm start, control will be returned
with the prompt for the current drive (eg, "A."). Command lines

may then be entered from the console keyboard. CDOS places another
jump instruction at locations 5, 6 and 7. The normal way to make
system requests of CDOS (those described below) is to call location 5.
The address stored at locations 6 and 7 is the address of the
beginning of CDOS znd thus marks the upper limit of user memory.

The following address map describes the memory area from 0 to 0FFH.
All addresses are in hex.

```
        0...2           CDOS re-entry
        3               I/O byte
        4               reserved
        5...7           system request call
        8...40          interrupt vectors
        40...5B         reserved
        5C...6B         default File Control Block 1 (FCB-1)
        6C...7B         default File Control Block 2 (FCB-2)
        7C...7F         reserved
        80...FF         default command-line buffer
```

When a .COM program is run by typing the program name on the
console, the default command-line buffer and default file control
blocks are used as follows. FCB-1 will contain the first filename,
if any, typed after the program name. FCB-2 will contain the
second filename, if any. The default buffer will contain the
entire command line following the program name. For example, if
this command line is typed:

        PROG FILE1.Z80 FILE2.COM

CDOS will place "FILE1Z8O" in FCB-1, "FILE2COM" in FCB-2,
" FILE1.Z80 FILE2.COM" in the command-line buffer, and load
and execute PROG.COM at 100H. Note that the second FCB starts
before the end of the first FCB. Before using FCB-1, FCB-2
should be moved. If it is not moved, part of FCB-1 will be
destroyed.

The command line which is placed in the default buffer can be
used to send more than two filenames to a program, or to start
execution of a program with various options specifed. For the
following command line:

        PROG FILE1.Z8O FILE2.COM OPTION1 OPTION2

the string of ASCII characters " FILE1.Z8O FILE2.COM OPTION1 OPTION2"
will be stored beginning at location 81H. The byte at location
80H will contain the length of the string. The byte following
the string will contain a null (00). PROG.COM can then look at
the command line stored in the default buffer to determine which
options were specified.

When a program is loaded, the disk buffer is set to 80H, which
is the default command buffer. If the disk is then read to or

written from, this buffer will be altered. The program must
either reset the disk buffer to another area or move the command
line before accessing the disk, if it is desired to save the
command line.

```
**************************************************
CHAPTER 2:  DEVICE I/O - LIST OF CDOS SYSTEM CALLS
**************************************************
```

CDOS has a set of system calls for device input and output. ALL
input and output should be done through these calls. This allows user
programs to be independent of physical devices. If a change needs to
be made in a device driver, it has only to be done once in the
system drivers. This chapter gives a detailed description of the
CDOS system calls. They are roughly divided into three sections:
the first section covers device I/O, where all devices are included
except disk drives. The next section covers the system calls
used to access disk files (disk I/O, opening and closing files,
etc.)   The last section covers several useful additional calls.
To use one of these routines the C register must be set to the
function number given below with the title of each instruction.
The other registers are set-up as that function requires (for
example the E or DE registers usually contain the parameter passed),
and a "CALL 5" instruction is executed. [Remember that CDOS
initializes location 5 with a jump instruction. This is done so
that the location of CDOS in memory is transparent to a user
program. A user program using the CDOS system functions does
therefore not need to do a CALL to a particular address in CDOS.]
For a complete summary of the CDOS system calls, refer to Chapter 3.
The system calls given below are in numerical order in each of the
three sections.

```
++++++++++++++++++++++++++
CDOS DEVICE FUNCTION CALLS
++++++++++++++++++++++++++
```

These system calls involve device I/O with all devices except disk
drives. The number given preceding each CDOS function is the number
which should be loaded into the C register prior to the "CALL 5".
The number is given first in hex and then in decimal in parentheses.

```
---------------------------
1 - READ CONSOLE (with echo)
---------------------------
```
This call is used to retrieve one byte from the console. The byte
will be returned in the A register. CDOS does not return to the
user program until a character is read and echoed back to the console.
The parity bit is set to 0. Note that a Control-Z (^Z) character is
usually considered an end of file mark.

```
----------------
2 - WRITE CONSOLE
----------------
```
This call is used to write one ASCII character to the console. The
character is placed in the E register before the call. CDOS will wait
until the console is ready to recieve the character and then print it.

After Control-P (^P) is typed all subsequent characters are sent
to both the console and the printer, until a second control-P is
typed (thus Control-P acts as a toggle switch). Control-W also
causes subsequent chacters to be sent to both the console and
the printer, and Control-T causes them to be sent only to the
console again. Control-W and Control-T are usually edited into a
file so that when that file is typed out on the console, it can stop
and start the printer at the appropriate places.

Control-I is the tab control. It is converted to spaces so that
the cursor is positioned at one of the standard tab stops, 1, 9,
17, 25, 33, 41,... However, the tab is still stored internally in
a file as the single ASCII character, 09H.

```
---------------
3 - READ READER
---------------
```
This call will read one character from the paper tape reader. All
8 bits are read. The character will be returned in the A register.
If it is the end-of-file character (Control-Z), the ZERO flag is set.

```
---------------
4 - WRITE PUNCH
---------------
```
This call will punch one character on the paper tape punch. All
8 bits are punched. The character is placed in the E register before
the call. CDOS will wait until the punch is ready to receive the
character.

```
--------------
5 - WRITE LIST
--------------
```
This call will print a character on the printer. Before the call,
the character to be printed is placed in the E-register. Tabs are
not expanded. CDOS will wait for the printer to accept the character
before it returns.

```
----------------
7 - GET I/O BYTE
----------------
```
For extra I/O devices, an "IOBYTE" has been provided. This byte
is not currently used by CDOS, but it is provided for the user's
programs. This function call returns the "IOBYTE" in the A register.
The format of the byte is:

```
        BIT    : 7 : 6 : 5 : 4 : 3 : 2 : 1 : 0 :
               : PRN : PUNCH   : READER  : CONSOLE    :
```

Thus up to eight consoles can be designated, four each of paper-tape
punch and reader, and one printer.

----
8 - SET I/O BYTE

This call allows the user program to set the "IOBYTE". The E register
contains the byte prior to the call. See above for the format of the
byte.

----
9 - PRINT BUFFER

This call will print a string of ASCII characters which has been
terminated with the "$" character. The DE register pair is set up
with the address of the beginning of the string before the call is
made to CDOS. If the printer toggle is on, the message will also
be sent to the printer.

----
10 (0AH) - INPUT BUFFERED LINE

This call will read an input line from the console. The DE register
must be pointing to an available buffer before the call is made to
CDOS. The first byte of the buffer must contain the maximum length
of the buffer. On return from this call the second byte of the
buffer will contain the actual length entered. The line that is
input will be stored beginning at the third byte. If the buffer
is not full, the byte at the end of the line will contain a zero.

When the line is being entered, the following characters will have
a special meaning:

        Control-C (^C)   Abort. Warm boot back to CDOS.
        Control-E (^E)   Physical CR-LF. The line is not
                              terminated and nothing is entered
                              into the buffer. This character is
                              used to enter a line longer than
                              can be printed on the console.
        Control-P (^P)   Toggle printer/console link. When this
                              character is first typed, the link is
                              toggled ON. All characters will then
                              be sent to the console and the printer.
                              The next time the character is typed,
                              the toggle will be turned off. All
                              characters will then be sent to the
                              console only.
        Control-R (^R)   Repeat what has been typed so far on
                              the line.
        Control-U (^U)   Delete the entered line and go back
                              to beginning of buffer for new line.
        Control-X (^X)   Delete the previous character and
                              echo the deleted character (used for
                              hard-copy terminals).
        RUBout   Delete the previous character and back

```
                               up the cursor (used for CRT terminals).
         DEL                   Same as RUBout.
         Underscore            Same as RUBout.
         Backspace (^H)        Same as RUBout.
```

```
-----------------------------
11 (OBH) - TEST CONSOLE READY
-----------------------------
```
The console is tested to see if a character has been typed. If a
character has been typed, 0FFH is returned in the A register. If no
character has been typed, 0 is returned in the A register.


```
-----------------------------------------
128 (80H) - READ CONSOLE (without echo)
-----------------------------------------
```
This call is the same as "READ CONSOLE (with echo)" except that
it does not echo the character after it is read. The byte is
returned in the A register.


```
------------------------------
142 (BEH) - SET CURSOR ADDRESS
------------------------------
```
This call will set the cursor at the specified address. This command
will only work when the console is a CRT with cursor addressing. The
D register is set up with the column address (1 through 80 for most
CRT's) and the E register is set up with the row address (1 through
24 for most CRT's).


```
+++++++++++++++++++++++++
CDOS DISK FUNCTION CALLS
+++++++++++++++++++++++++
```

CDOS divides the disk into regions called files. Files are
referenced through file control blocks (FCBs). FCBs are 33 bytes
long and have the following format, where each of the numbers below
stands for one byte:

```
FCBDK   Disk descriptor         0       (0=current disk, 1=drive-A,
                                           2=B, 3=C, 4=D)
FCBFN   File name               1...8   (right-filled with blanks)
FCBFT   File type (extension)   9...11  (right-filled with blanks)
FCBEX   File extent             12      (initially 0; is incremented
                                           by one in every new
                                           extent of 16 Kbytes)
        Reserved                13...14
FCBRC   Record count            15      (total number of 128-byte
                                           sectors or records)
FCBMP   Cluster allocation map  16...31 (allocated clusters 2
                                           through 240)
FCBNR   Next record             32      (next record to be read or
                                           written; has the value
                                           0 through 127)
```

It should be noted that directory entries on the disk consist of

32-byte FCBs. The last byte, FCBNR, which points to the next record,
is omitted.

----
12 (0CH) - DESELECT CURRENT DISK

The current disk is deselected. The CDOS disk driver can be changed
to perform any desired function at this time to deselect the disk.
Currently the driver outputs a 0 to part 34H when this function is
selected.

----
13 (ODH) - RESET CDOS AND SELECT DRIVE A

CDOS is initialized, all disks are logged-off, and drive A is
selected as the current drive. The other disks will be logged-on
again as soon as they are accessed.

----
14 (0EH) - SELECT DISK DRIVE

The disk drive number in the A register is selected as the current
disk. The drive number in the A register is 0 for drive A, 1 for
drive B, 2 for drive C, or 3 for drive D.

----
15 (0FH) - OPEN DISK FILE

The FCB pointed to by the DE register pair is opened to allow reading
or writing to the file whose name is specified in the FCB. The A
register returns with -1 (OFFH or 255D) if the file is not found,
or the directory block number if the file is found. Block numbers
start at 0 and there is one block number for every four directory
entries. The HL register pair returns pointing to the directory
entry in memory.

----
16 (10H) - CLOSE FILE

The FCB pointed to by the DE register pair is closed and the disk
directory is updated. The file described by the FCB must have been
previously opened or created; if it has not been, an unpredictable
directory entry will be written to the disk. A file to which bytes
have just been written MUST be closed using this function or the
entire last extent will be unable to be read.

----
17 (11H) - SEARCH DIRECTORY

The directory is searched for the first occurrence of the file
specified in the FCB pointed to by the DE register pair. ASCII "?"
(3FH) in the FCB matches any character. The block number (see
description of directory block numbers in 0FH - Open Disk File, above)
is returned in the A register if found, if the file is not found, -1
(0FFH or 255D) is returned in A. HL is returned pointing to the

directory entry in memory. An important point to note about this
call and the one following (12H) is that they will get the directory
entry whether it has been erased or not; ie, these calls do not check
to see if a file has been erased. Files are erased by placing a 0E5H
in the first byte (FCBDK); the rest of the FCB is left unchanged.

----
18 (12H) - FIND NEXT DICTORY ENTRY

This call is the same as 11H (17) above except that it finds the
NEXT occurrence of the filename in the directory. This may be either
the next extent of a file occupying more than one extent, or another
filename if the match-character, "?", was used in the FCB. This
call is made after function 17 and no other disk system call can be
made between these calls.

----
19 (13H) - DELETE FILE

The file specified by the FCB pointed to by the DE register pair is
deleted from the disk directory. ASCII "?" in the FCB matches any
character. The number of directory entries deleted is returned in the
A register.

----
20 (14H) - READ NEXT RECORD

The DE register pair points to a successfully OPENED FCB. The next
record (128 bytes) is read into the current disk buffer. The FCBNR in
the FCB is incremented to read the next record. One of the following
codes is returned in the A register:

        0 - read completed
        1 - end of file
        2 - read attempted on unwritten cluster
                (random access file only)

----
21 (15H) - WRITE NEXT RECORD

The DE register pair points to a successfully OPENED FCB. The next
record (128 bytes) is written into the file from the current disk
buffer. The FCBNR in the FCB is incremented to be ready to write
the next record. One of the following codes is returned in the
A register:

        0 - write completed
        1 - extent error (attempted to close an unopened extent)
        2 - out of disk space (limited to 81K - small, 241K - large)
       -1 (0FFH or 255D)
          - out of directory space (limited to 64 extents)

----
22 (16H) - CREATE FILE

The file specified in the FCB pointed to by the DE register pair is
created on the disk. The A register is returned containing the
block number of the directory entry (see 0FH - Open Disk File), or
-1 (0FFH or 255) if no more directory space is available.

----
23 (17H) - RENAME FILE

This call will rename a disk file. The DE register pair points to
the FCB to be renamed. The old file name and file type are in the
first 16 bytes and the new file name and file type are in the second
16 bytes of the FCB. ASCII "?" in the FCB will match with any
character. The A register returns containing the number of directory
entries renamed.

----
24 (18H) - DISK LOG-IN VECTOR

The A register is returned, specifying the disks that are logged-in.
Each bit represents one disk drive logged-in. If the bit is a one,
then it is logged-in; else it is offline. The least significant bit
is the A drive, next most significant (Bit 1) is drive B, etc. Since
there would be no more than four drives, the upper four bits are 0's.

----
25 (19H) - CURRENT DISK

The number of the current disk drive is returned in the A register.
0 = drive A, 1 = drive B, 2 = drive C, 3 = drive D.

----
26 (1AH) - SET DISK BUFFER

The buffer pointed to by the DE register pair is used for disk I/O.
When a program is loaded, the disk buffer is initially located at 80H.

----
27 (1BH) - DISK CLUSTER ALLOCATION MAP

The BC register pair returns pointing to a bit map that corresponds
to the allocated clusters on the disk. The DE register pair returns
containing the capacity of the current disk in number of clusters.
The A register returns containing the number of records or sectors
per cluster (8). This system call is used by "STAT".

----
131 (83H) - READ LOGICAL BLOCK

This system call will read a logical block from the disk without any
attention to the files it may contain (ie, no FCB is specified). A
block is defined to be one sector or record of 128 bytes. When this
function is called, the DE register pair should contain the block
number and the B register should contain the disk number (0 for
current drive, 1-4 for A-D). The high bit of the B register contains
a 1 for an interleaved and a 0 for a non-interleaved read. Interleaved

means the block which is read is found in the order CDOS stores it
(every fifth sector for small disks and every sixth sector for large
disks). Non-interleaved means the block which is read is found in
sequential order, the order it is physically stored on the disk.
The A register is returned with the status of the read according to
the following:

        0 - OK
        1 - I/0 error
        2 - illegal request
        3 - illegal block

An example will help to illustrate these points. CDOS makes use of
716 sectors on the small floppy disks. Therefore, the block numbers
which could legally he loaded into the DE register are 0 through
715 decimal, or 0 through 2CBH. Suppose that DE is loaded with the
Value 2 and the B register with 0 (current disk, noti-interleaved read).
Thus, since the sectors are numbered beginning with 1, sector 3 would
be read into memory in the disk buffer (located at 80H if it has not
been changed). The same read with the B register loaded with 80H
(current disk, interleaved read) would read sector OBH (the third
sector when they are read every fifth one).

----
132 (84H) - WRITE LOGICAL BLOCK

This system call will write a logical block or sector to the disk
without any attention to the file there (no FCB is specified). The
registers are set up and returned in the same way as they were for
the Read Logical Block system call above.

----
134 (86H) - FORMAT NAME TO FCB

This system call will build a File Control Block. The HL register
pair points to the start of the input line. The DE register points
to the place in memory where the FCB is to be built. The input
line is of the format:

        d:filename.ext

where d stands for one of A-D, the filename is up to 8 letters with
a 3-letter extension. The FCB is then built from this input line,
converting lower case to upper case. The input line is terminated
by an ASCII "/" or any character less than 21H (such as a space or
carriage return).

On return the HL register pair points to the terminator that ended
the build operation. The DE register pair points to the start of the
new FCB.

----
135 (87H) - UPDATE DIRECTORY ENTRY

The last disk I/O function called must have been function 17 or 18,

Search Directory or Find Next Entry. The DE register pair points
to the FCB used in the function call 17 or 18. The directory entry
is then updated on the disk; this means that the entry is written
back to the disk without the user having to specify a block. The
user merely specifies a filename when calling 17 or 18. This is
useful if it is desired to change a directory entry and write it
back to the disk.

----
139 (8BH) - HOME DISK

The disk drive specified in the B register (0 for current drive and
1-4 for drives A-D) is sent a command to "home" the head. The disk
drive head will return to track 0.

----
140 (BCH) - EJECT DISK

This call will eject the disk whose number is given in the E register
(0 for current drive and 1-4 for drives A-D, respectively), only if
the disk drive is a CROMEMCO Dual Disk Drive System, Model PFD with
the eject option. Otherwise, the call will have no effect.

++++++++++++++++++++++
ADDITIONAL SYSTEM CALLS
++++++++++++++++++++++

Several additional CDOS system calls have been added for the pro-
grammer's convenience. These calls are explained in this section.

----
0 - ABORT

This call will abort the current program and return control to CDOS.
This call has the same effect as jumping to location 0.

----
129 (81H) - GET USER REGISTER POINTER

This call is provided for future expansion of CDOS to a multipro-
gramming system. The BC register pair returns pointing to the
user register pointers.

----
130 (82H) - SET USER CONTROL-C ABORT

When Control-C (^C) is typed, the system usually aborts and returns
control to CDOS. This call allows the programmer to assign an
address to which to jump when Control-C is typed (ie, users can
assign their own function to Control-C). The address is given in
the DE register pair. Note that if DE contains a zero, the system
abort is reset. Jumping to location 0 at any time still causes a
return to CDOS, also with the Control-C being restored to its
original function.

```
----------------------------
136 (88H) - LINK TO PROGRAM
----------------------------
```
This enables one command program to call another. The default
command-line buffer and default FCBs for the new program must be
set up prior to this call if that program expects to be able to
use them. The DE register pair should contain the address of the

FCB of the new program (which must have an extension of ".COM").
If the new program is NOT found, the A register returns containing
-1 (0FFH or 255); also in this case the first 80H bytes (from 100H to
17FH) will be destroyed because this is used in reading the directory.
Otherwise, execution begins at 100H and no return is made to the
original program.

```
--------------------
137 (89H) - MULTIPLY
--------------------
```
This system call provides a 16-bit multiply. The HL and DE register
pairs contain the two 16-bit factors, and the answer is returned in
register DE (ie, DE = DE*HL).

```
------------------
138 (8AH) - DIVIDE
------------------
```
This system call provides a 16-bit divide. The HL register pair
should contain the dividend, and the DE register pair, the divisor.
The quotient is returned in HL, and the remainder in DE (ie,
HL = HL/DE with DE = remainder). DE contains the remainder.

```
-------------------------------
141 (8DH) - GET VERISION NUMBER
-------------------------------
```
This call will return the version-number of CDOS in the B register
and the release-number in the C register.

```
******************************************
CHAPTER 3:  SUMMARY OF CDOS FUNCTION CALLS
******************************************
```

Following is a summary table listing all the system calls described
in Chapter 2 along with their entry and return parameters. The
functions are listed in numerical order, ie, by order of the number
which is loaded into the C register to achieve the desired function.

NUMBER	FUNCTION	ENTRY PARAMETERS	RETURN PARAMETERS
0	ABORT	none	none

```
1          READ CONSOLE      none                A = character
           (with echo)
2          WRITE CONSOLE     E = character       none
3          READ READER       none                A = character
                                                 Z flag set = end of file
4          WRITE PUNCH       E = character       none
5          WRITE LIST        E = character       none
7          GET I/O BYTE      none                A = I/O byte
8          SET I/O BYTE      E = I/O byte        none
9          PRINT BUFFER      DE = buffer address none
10 (0AH)   INPUT BUFFERED    DE = buffer address none
           LINE
11 (0BH)   TEST CONSOLE      none                A = OFFH (-1) if ready
           READY                                 A = 0 if not ready
12 (0CH)   DESELECT          none                none
           CURRENT DISK
13 (ODH)   RESET CDOS AND    none                none
           SELECT DRIVE A
14 (OEH)   SELECT DISK       E = disk drive      none
15 (0FH)   OPEN DISK FILE    DE = FCB address    A = directory block
                                                 A = OFFH if not found
16 (10H)   CLOSE FILE        DE = FCB address    none
17 (11H)   SEARCH            DE = FCB address    A = directory block
           DIRECTORY                             A = 0FFH if not found
18 (12H)   FILE NEXT ENTRY   DE = FCB address    A = directory block
                                                 A = 0FFH if not found
19 (13H)   DELETE FILE       DE = FCB address    A = number of entries
                                                     deleted
20 (14H)   READ NEXT         DE = FCB address    A = 0 if ok
           RECORD                                A = 1 if end of file
                                                 A = 2 if tried to read
                                                     unwritten records
21 (15H)   WRITE NEXT        DE = FCB address    A = 0 if ok
           RECORD                                A = 1 if extent error
                                                 A = 2 if out of disk
                                                     space
                                                 A = -1 (0FFH) if out
                                                     of directory space
```

```
NUMBER    FUNCTION          ENTRY PARAMETERS    RETURN PARAMETERS
------------------------------------------------------------------
22 (16H)  CREATE FILE       DE = FCB address    A = directory block
                                                A = 0FFH if not found
23 (17H)  RENAME FILE       DE = FCB address    A = number of entries
                                                    renamed
24 (18H)  DISK LOG-IN       none                A = those disks
          VECTOR                                    logged-in
25 (19H)  CURRENT DISK      none                A = disk number
26 (1AH)  SET DISK BUFFER   DE = buffer address none
27 (lBH)  DISK CLUSTER      none                BC = address of bitmip
          ALLOCATION MAP                        DE = number of clusters
                                                A = sectors/cluster
128 (80H) READ CONSOLE      none                A = character
           (with no echo)
```

```
129 (81H) GET USER         none              BC = pointer to user
          REGISTER                              register
          POINTER                               pointerrs
130 (82H) SET USER         DE = address      none
          CONTROL-C
          ABORT
131 (83H) READ LOGICAL     DE = block number A = 0 if ok
          BLOCK            B = disk number   A = 1 if I/O error
                           B top bit = 1 if  A = 2 if illegal request
                              interleaved    A = 3 if illegal block
132 (84H) WRITE LOGICAL    DE = block number A = 0 if ok
          BLOCK            B = disk number   A = 1 if I/O error
                           B top bit = 1 if  A = 2 if illegal request
                              interleaved    A = 3 if illegal block
134 (86H) FORMAT NAME      HL = address of   HL = address of
           TO FCB              string             terminator
                           DE = FCB address  DE = FCB address
135 (87H) UPDATE           DE = FCB address  none
          DIRECTORY ENTRY
136 (88H) LINK TO PROGRAM  DE = FCB address  A = -1 if error, else
                                             execute at 100H
137 (89H) MULTIPLY         DE = factor 1     DE = product
                           HL = factor 2
138 (8AH) DIVIDE           HL = dividend     HL = quotient
                           DE = divisor      DE = remainder
139 (8BH) HOME DISK        B = disk number   none
140 (8CH) EJECT DISK       E = disk number   none
141 (8DH) GET VERSION      none              B = version-number
                                             C = release-number
142 (8EH) SET CURSOR       D = column address none
          ADDRESS          E = row address
```

                                              Page 109

```
*****************************************
CHAPTER 1:  ROUTINES AVAILABLE IN ASMLIB
*****************************************
```

The library file "ASMLIB.REL" has been provided for your use in
assembly language programming. There are three types of routines
(decimal conversion, hexadecimal conversion, and character I/O).
An example of how to add these routines to your program follows.

```
        LINK PROG,ASMLIB/S/G
```

This example will load a program called "PROG" and then load only
the routines in "ASMLIB" that are required. See Part II on LINK for
more information.

```
+++++++++++++++++
DECIMAL CONVERSION
+++++++++++++++++
```

```
------------------------------------
ADEC - DECIMAL TO BINARY CONVERSION
------------------------------------
```

This routine will convert a decimal string to a binary number.
The following example will illustrate how to use this routine.

```
        LD      BC,STRING       ;point to ASCII string
        CALL    ADEC            ;convert to binary
```

The routine will return with the HL register pair containing the
16-bit binary number and the BC register pair pointing to the first
non-digit.

```
-------------------------------------------------------
BINDF, BINDB, BINDS, BIND - CONVERT BINARY TO DECIMAL
-------------------------------------------------------
```

These routines will convert a binary number into a decimal string.
The routine "BINDF" will zero fill, "BINDB" will fill with spaces,
"BINDS" will suppress printing of leading zeros, and "BIND" will fill
with the character in the A register. In the following example leading
zeros will be printed as "*"s.

```
        LD      HL,STRING       ;store ASCII string here
        LD      BC,(BINARY)     ;this is binary number
        LD      A,'*'           ;fill character
        CALL    BIND            ;convert to ASCII string
```

Six bytes most be reserved for the string, unless "BINDS" is used,

in which case this routine will use only the number of bytes that are
not leading zeros.

```
*********************
HEXADECIMAL CONVERSI0N
*********************
```

```
------------------------------
AHEX - ASCII TO HEX CONVERSION
------------------------------
```

     This routine will convert a hexadecimal string to a binary number.
The calling sequence is

```
        LD      BC,STRING       ;point to ASCII string
        CALL    AHEX            ;convert to binary
```

     The routine will return with the HL register pair containing the
binary number and the BC register pair pointing to the first non-
hexadecimal digit.

```
------------------------------
BINH4 - BINARY TO 4 HEX DIGITS
------------------------------
```

     This routine will convert the binary number in the BC register
pair to 4 ASCII digits. The calling sequence is

```
        LD      BC,(NUMBER)     ;get binary number
        LD      HL,STRING       ;store ASCII String here
        CALL    BINH4           ;convert to ASCII
```

```
------------------------------
BINH2 - BINARY TO 2 HEX DIGITS
------------------------------
```

     This routine will store 2 ASCII digits. The calling sequence is

```
        LD      A,(NUMBER)      ;get binary number
        LD      HL,STRING       ;store ASCII string here
        CALL    BINH2
```

```
------------------------------
BINH1 - BINARY TO 1 HEX DIGIT
------------------------------
```

     This routine will store 1 ASCII digit. The calling sequence is

```
        LD      A,(DIGIT)       ;get binary digit (lower 4 bits of A)
        LD      HL,STRING       ;store digit here
        CALL    BINH1
```

```
++++++++++++++++++++++
CHARACTER I/O ROUTINES
++++++++++++++++++++++
```

     Providing character I/O which is device independent adds considerable
power to a program. These routines allow opening a file by symbolic name
(disk or device) and then calling the same routines for I/O. There are
routines for both ASCII and BINARY data. The binary calls pass 8 bits of
data. The ASCII calls pass only printable characters plus carriage return,
line feed, and tab. All other characters are passed as two characters
(an up arrow and the corresponding printable character; eg, Control-B
would be printed as "^B"). Devices are referenced by using the following
symbolic names; all others are considered disk files.

```
          RDR:[#]  -  reader (0..4)
          PUN:[#]  -  punch  (0..4)
          LST:[#]  -  printer (0..1)
          PRT:[#]  -  printer (0..1)
          CON:[#]  -  console (O..7)
          DUM:     -  dummy
```

     The option number is set into the "IOBYTE" to select device units.
The symbolic name "DUM:" is used to throw away output, or as end of file.

     An extended FCB (XFCB) is used which includes character pointers.
When the XFCB is initialized, the number of buffers are specified (each
is 128 bytes). Only disk files use the buffers.

     The format of the XFCB follows.

name	position	length	description
ZCNT	0	1	byte count (O..127 or 255)
ZFLG	1	1	flags
ZFCB	2..34	33	CDOS file control block (FCB)
ZBPTR	35..36	2	buffer pointer (1st buffer)
ZBCUR	37	1	current buffer
ZNBUF	38	1	number of buffers
ZFBUF	39	1	full number of buffers
	40		total length

     The byte count indicates a non-disk device if it contains 255. ZFLG
will then contain the system call for that device. The followino are the
flags.

```
          RDR:    3
          PUN:    4
          LST:    5
          PRT:    5
          CON:    1
          DUM:    0
```

     The initial format of an XFCB should be

```
          DEFB     0
          DEFS     34
          DEFW     buffer address,0
          DEFB     number of buffers
```

FNAME - SET UP XFCB

     This routine sets up an XFCD from an FCB. If the routine is called
with the A register equal to 0, then the extension in the FCB is used.
If the A register is not equal to 0, then the A, B, and C registers
contain the extension that is to be used. The example below will set up
an XFCB from the system FCB at location 5CH with an implied extension
of ".PRN".

```
          LD       HL,5CH          ;point to system FCB
          LD       DE,XFCB1        ;point to XFCB
          LD       A,'P'           ;".PRN" extension
          LD       BC,'RN'
          CALL     FNAME           ;build XFCB
```

XDISK - SET UP SPECIAL XFCB

     This routine will modify an XFCB using a letter in the A register.
If the A register contains A through W, this is considered to be a disk
identifier. If the A register contains "X", the XFCB is converted to use
the console. If it contains a "Y", the XFCB is converted to use the list
device. If it contains a "Z", then the XFCB is converted to use the dummy
driver. This routine allows the decoding of parameters such as the
assembler uses for its files. In the following example the XFCB is
converted to use the console.

```
          LD       DE,XFCB         ;point to the XFCB
          LD       A,'X'           ;make it the console
          CALL     XDISK           ;convert XFCB
```

ZNEW - OPEN NEW XFCB

     This routine will delete any old file with the same name and then
create and open a new file. If there is an error the ZERO flag is set
and the HL register pair points to an error message. In the following
example a new file is created.

```
          LD       DE,XFCB         ;point to XFCB
          CALL     ZNEW            ;create a new file
          CALL     Z,ZIOER         ;print error and abort
```

                                                     Page 114

ZOPN - OPEN OLD XFCB

     This routine will open an existing file. If there is an error the
ZERO flag is set and the HL register pair points to an error message.
In the following example an old file is opened.

          LD   DE,XFCB   ;point to XFCB
          CALL   ZOPN   ;open the file
          CALL   Z,ZIOER   ;print error and abort

ZCLOS - CLOSE XFCB

     This routine will close a file. In the follwing example a file is
closed.

          LD   DE,XFCB   ;point to XFCB
          CALL   ZCLOS   ;close the file
          CALL   Z,ZIOER   ;print error and abort

PCHAR - PUT CHARACTER (BINARY)

     This routine is used to output binary characters. In the following
example a character is output.

          LD   DE,XFCB   ;point to XFCB
          LD   C,(HL)   ;get character to output
          CALL   PCHAR   ;output character
          CALL   Z,ZIOER   ;print error and abort

PUTC - PUT CHARACTER (ASCII)

     This routine is used to output ASCII characters to a disk file or
a device such as the console, a printer, etc. In the following example
a character is output.

          LD   DE,XFCB   ;point to XFCB
          LD   C,(HL)   ;get character to output
          CALL   PUTC   ;output character
          CALL   Z,ZIOER   ;print error and abort

                                                  Page 115

GCHAR - GET A CHARACTER

     This routine is used to input characters from a disk file or a
device. In the following example, a character is returned in the A
register.

          LD   DE,XFCB   ;point to XFCB

```
            CALL      GCHAR               ;get a character
            CP        1AH                 ;Q, end of file
            JP        Z,EOF               ;yes, end of file
```

     When an unwritten random record is read, it is treated as an end
of file.

ZIOER - PRINT FILE ERROR MESSAGE

     This routine is the standard error routine. When an error occurs
in one of the file handling routines, the HL register pair points to
the error message, the DE register pair points to the XFCB, and the
ZERO flag is set. This allows the instruction "CALL Z,ZIOER" to follow
a disk handling routine. In the following example, a character is written.
If there is an error, it will be printed and control will be passed to
CDOS.

```
            LD        DE,XFCB             ;point to XFCB
            LD        C,(HL)              ;get a character
            CALL      PUTC                ;output character
            CALL      Z,ZIOER             ;print error and abort
```

PFNAM - GET FILE NAME FOR PRINTING

     This routine will extract the file name from the XFCB and form a
printable string. The string will be in the following format

          d:filname.ext

     Where d: is an optional disk number (A-D), filename is the name
of the user file (1 to 8 characters), and ext is the filename extension
(0 to 3 characters). The string is terminated by a byte equal to zero.
The length of the string is returned in the A register. In the following
example a string is formed from the XFCB.

```
            LD        DE,XFCB             ;point to XFCB
            LD        HL,BFLINE           ;store string here
            CALL      PFNAM               ;form string
            CALL      PRNT                ;print the file name
```

                                                  Page 116

PRNT - PRINT A LINE

     This routine will print a string which ends with either a zero-byte
or a carriage return. If a carriage return is found, the carriage return
and a line feed is output. In the following example the string
"THIS IS A STRING" is output.

```
        LD      HL,STRING       ;point to string
        CALL    PRNT            ;print the string
         :
         :
STRING: DEFB    'THIS IS A STRING',0


---------------------------
ABORT - ABORT USER PROGRAM
---------------------------

     This routine will print a message and then abort to CDOS.  The
format of the message is the same as in the previous example.  In the
following example the message "*** END OF JOB ***" is output to the
console and control is returned to CDOS.

        LD      HL,STRING       ;point to string
        CALL    ABORT           ;abort program
         :
         :
STRING: DEFB    '*** END OF JOB ***',13
```

                                                       Page 117


```
**********************
CHAPTER 2:  AN EXAMPLE
**********************
```

     The program "EXAMPLE.Z80" has been included as an example. To run
this example use the batch file "EXAMPLE.CMD". The first line of the
example is typed by the user. The rest of the example is typed by the
computer.

```
B.@ EXAMPLE
BATCH  VERSION 00.02

B.ASMB EXAMPLE.AAX
CROMEMCO CDOS Z80 ASSEMBLER version 02.02
```

                                                       Page 118


```
CROMEMCO CDOS Z8O ASSEMBLER version 02.02                 PAGE 0001
*** EXAMPLE ***

                  0002 ;
                  0003 ;THIS PROGRAM WILL INPUT FROM ONE
                  0004 ;DISK FILE OR DEVICE
                  0005 ;AND OUTPUT TO ONE DISK FILE OR DEVICE
                  0006 ;
                  0007 ;TO CALL THIS PROGRAM TYPE
                  0008 ;"EXAMPLE filenam1.ext filenam2.ext" where
                  0009 ;"filenam1.ext" IS THE OUTPUT FILE/DEVICE and
                  0010 ;"filenam2.ext" IS THE  INPUT FILE/DEVICE
                  0011 ;
                  0012         NAME    EXAMPL
```

```
                0013          EXT     FNAME        ;SET UP XFCB
                0014          EXT     ZNEW         ;OPEN NEW XFCB
                0015          EXT     ZOPN         ;OPEN OLD XFCB
                0016          EXT     ZCLOS        ;CLOSE XFCB
                0017          EXT     ZIOER        ;ERROR ROUTINE
                0018          EXT     ABORT        ;END PROGRAM
                0019          EXT     GCHAR        ;GET A CHARACTER
                0020          EXT     PUTC         ;PUT A CHARACTER
                0021 ;
                0022 ;START OF PROGRAM
                0023 ;
0000' 3A5D00    0024 START:   LD      A,(5DH)      ;1ST BYTE OF FILNAME
0003' FE20      0025          CP      ' '          ;Q, BLANK FILE NAME
0005' CA6500'   0026          JP      Z,ERROUT     ;YES, ERROR
0008' 97        0027          SUB     A            ;USE EXT FROM FCB
0009' 215COO    0028          LD      HL,5CH       ;POINT TO 1ST FCB
000C' 117FOO'   0029          LD      DE,OXFCB     ;POINT TO OUTPUT XFCB
000F' CD0OOO#   0030          CALL    FNAME        ;BUILD XFCB
0012' CD0000#   0031          CALL    ZNEW         ;CREATE A NEW FILE
0015' CC0000#   0032          CALL    Z,ZIOER      ;ERROR
                0033 ;
0018' 3A6D00    0034          LD      A,(6DH)      ;1ST BYTE OF FILNAME
001B' FE20      0035          CP      ' '          ;Q, BLANK FILE NAME
001D' CA6500'   0036          JP      Z,ERROUT     ;YES, ERROR
0020' 97        0037          SUB     A            ;USE EXT FROM FCB
0021' 216COO    0038          LD      HL,6CH       ;POINT TO 2nd FCB
0024' 11A700'   0039          LD      DE,IXFCB     ;POINT TO INPUT XFCB
0027' CD1000#   0040          CALL    FNAME        ;BUILD XFCB
002A' CD0000#   0041          CALL    ZOPN         ;OPEN OLD XFCB
002D' CC1600#   0042          CALL    Z,ZIOER      ;ERROR
                0043 ;
0030' 11A700'   0044 LOOP:    LD      DE,IXFCB     ;POINT TO INPUT XFCB
0033' CD0000#   0045          CALL    GCHAR        ;GET A CHARACTER
0036' FE1A      0046          CP      1AH          ;Q, END OF FILE
0038' 280C      0047          JR      Z,EOF        ;YES
003A' 117FOO'   0048          LD      DE,OXFCB     ;POINT TO OUTPUT FCB
003D' 4F        0049          LD      C,A          ;GET CHARACTER
003E' CDOOOO#   0050          CALL    PUTC         ;PUT ASCII CHARACTER
0041' CC2E00#   0051          CALL    Z,ZIOER      ;ERROR
0044' 18EA      0052          JR      LOOP         ;GET NEXT CHARACTER
                0053 ;
0046' 117F00'   0054 EOF:     LD      DE,OXFCB     ;CLOSE OUTPUT XFCB
0049' CDO000#   0055          CALL    ZCLOS
004C' 215200'   0056          LD      HL,EOFMSG    ;POINT TO EOF MESSAGE
004F' CDOOOO#   0057          CALL    ABORT        ;ABORT PROGRAM
                0058 ;
```

CROMEMCO CDOS Z8O ASSEMBLER version 02.02   PAGE 0002
*** EXAMPLE ***

```
0052' 2A2A2A20  0059 EOFMSG   DEFB    '*** END OF JOB ***',13
      454E4420
      4F46204A
      4F42202A
```

    The program "EXAMPLE.COM" is now ready to be executed. To use the
program type in the name of the program followed by an output file and
an input file. For example

        B.EXAMPLE NEWFILE.Z80 EXAMPLE.Z80

    This example will copy the file "EXAMPLE.Z80" to the file
"NEWFILE.Z80".
    Device names may also be used. The following example will type the
file "EXAMPLE.Z80" on the console.

        B.EXAMPLE CON: EXAMPLE.Z80

```
**********************************************************
PART VI - CREATING A NEW LUN TABLE FOR CROMEMCO FORTRAN IV
**********************************************************
```

```
**************************************************
PROCEDURE FOR CREATING A NEW LUN TABLE FOR FORTRAN
**************************************************
```

There have been a number of requests among our customers for information
on how to change the driver dispatch table (LUN Table) to accommodate
other I/O drivers with CROMEMCO FORTRAN IV. The purpose of this section
is to explain the method for doing this. The present LUN Table is
located in the FORTRAN Library file, FORLIB.REL, under the name: $LUNTB.

The Linker automatically searches FORLIB when linking FORTRAN programs
to satisfy any undefined symbols. LINK then loads these needed routines
into memory. However, if the LUN Table were defined PRIOR to the search
of FORLIB, the Linker would not load $LUNTB from FORLIB. This is done
by first composing the new LUN Table giving it the same name ($LUNTB),
then assembling it using ASMB, and finally linking it prior to the link
of FORLIB. This procedure is demonstrated below. However, first here
is a duplicate of the LUN Table which is presently used in CROMEMCO
FORTRAN:

```
              ENTRY    $LUNTB
              EXT      $DRV3,LPTDRV,DSKDRV
      $LUNTB: DB       0BH
      ONE:    DW       $DRV3
      TWO:    DW       LPTDRV
      THREE:  DW       $DRV3
      FOUR:   DW       $DRV3
      FIVE:   DW       $DRV3
      SIX:    DW       DSKDRV
      SEVEN:  DW       DSKDRV
      EIGHT:  DW       DSKDRV
      NINE:   DW       DSKDRV
      TEN:    DW       DSKDRV
              END
```

Note the use of the ENTRY statement to define the module. The symbols
$DRV3, LPTDRV, and DSKDRV stand for the console driver, line-printer
driver, and disk driver modules, respectively. The labels ONE through
TEN are provided for convenient reference; they mark the drivcr-address
which stands for each of the LUNs 1 through 10. As can be seen from
the above, LUNs 1 and 3-5 are presently assigned to the console, LUN 2
is assigned to the printer, and LUNs 6-10 are assigned to disk files.
(See FORTRAN IV Instruction Manual, Appendix B and page 15 for more
information on Logical Unit Numbers.) Also note in the above that the
first byte of the module (DB 0BH) must be one more than the maximum
LUN (in this case 10). Hence, more LUNs could be defined simply by
adding DW statements and by changing this first byte.

The present LUNs can be changed simply by rearranging the driver
addresses in each DW statement above. (LUN 3 should be preserved as
the console driver, however, as that is the one used by the system to
print out error messages.) Users may also write their own drivers in
Z-80 assembly code, assemble them with ASMB, and link them with the new
$LUNTB. To illustrate these ideas here is a sample altered LUN Table:

```
              ENTRY    $LUNTB
              EXT      $DRV3,LPTDRV,DSKDRV,SPTDRV
      $LUNTB: DB       21
      ONE:    DW       $DRV3
      TWO:    DW       $DRV3
      THREE:  DW       $DRV3
      FOUR:   DW       LPTDRV
      FIVE:   DW       LPTDRV
      SIX:    DW       SPTDRV
      SEVEN:  DW       SPTDRV
```

```
        EIGHT:  DW        DSKDRV
        NINE:   DW        DSKDRV
        TEN:    DW        DSKDRV
        ELEVN:  DW        DSKDRV
        TWELV:  DW        DSKDRV
        THIRTN: DW        DSKDRV
        FOURTN: DW        DSKDRV
        FIFTN:  DW        DSKDRV
        SIXTN:  DW        DSKDRV
        SEVNTN: DW        DSKDRV
        EIGHTN: DW        DSKDRV
        NINETN: DW        DSKDRV
        TWENTY: DW        DSKDRV
                END
```

In this example the user has added an EXTernal declaration for a serial
line-printer, SPTDRV. The LUN assignments have also been changed as
follows: LUNs 1 through 3 are assigned to the console, 4 and 5 are
assigned to the parallel-port printer, 6 and 7 are assigned to the
serial-port printer, 8 through 10 remain assigned to disk files, and
LUNs 11 through 20 have also been assigned to disk files.

The driver for the serial printer should be of the format:

```
                ENTRY    SPTDRV
        START:  ...
                :
                :
                END
```

The LUN file which has been created can now be assembled using ASMB
simply by typing:

```
        ASMB LUNTBNEW
```

where LUNTBNEW.Z80 is the name of this file on the disk. The source
file for the added driver (SPTDRIVR.Z80) must also be assembled; ASMB
will create .REL files for both these modules. These two files can
finally be linked to the FORTRAN by typing:

```
        LINK FORPROG,LUNTBNEW,SPTDRIVR
```

where FORPROG is the user's previously-compiled FORTRAN IV program.
LINK will automatically search FORLIB, but will ignore the $LUNTB file
there because LUNTBNEW was linked first. Note that the ENTRY statement
for LUNTBNEW.Z80 must have the same name as the original module ($LUNTB).