Xitan, Inc.
Disk BASIC Version 1.06

Preliminary Update Documentation

Written by Neil J. Colvin
June 16, 1978

Xitan Disk Basic Version 1.06
Preliminary Update Documentation

1. Program Text Inputting and Editing

A number of changes and additions have been made to the facilities provided for the manipulation of the program text.

1-1. Keyboard Input

The BACKSPACE key is now recognized as a character delete in addition to the DEL key. The key echos as BS-SPACE-BS. This facilitates keyboard input on a video device which allows backspacing and overwriting.

1-2. AUTO Command

The AUTO command has been enhanced to provide better user control and flexibility. The format of the command is:

```
AUTO [<starting line>[+]][,<increment>]
```

The only change in this format is the + option. The use of the + option specifies that the increment is to be added to the starting line before the first line number is generated.

If any line number generated by the AUTO command corresponds to an already existing line in the program, the line number is preceeded by a "+" when displayed on the terminal. To avoid overwriting the existing line, the entry of JUST the RETURN key will advance the line number to the next line without changing the exisitng one.

This automatic skipping of line numbers while in AUTO mode may be used at any time. Previously, the entry of just the RETURN key terminated the AUTO mode. Because of this new feature, a new AUTO mode termination has been provided. To terminate AUTO mode, enter CTL-E. This will immediately return to command mode, regardless of any other text entered on the same line.

1-3. The "." Line Number

To facilitate program entry and editing, a shorthand notation for the "current line" is provided. From command mode (ONLY), a period (".") may be substituted anywhere a line number is normally used. This period represents the last program line accessed by a program editing or execution command. For example:

```
10 A$=MID$(B$,5,6
EDIT .
10
```

Note that the EDIT command references line 10, the last line accessed.

After any execution error, the period is set to the line the error occured on, so that a

```
LIST .
```

will list the line in error.

When AUTO mode is terminated by a CTL-E, the period is set to the last line entered. To resume input with the next line, the command:

```
AUTO .+
```

may be used (see AUTO command, Section 1-2).

1-4. FIND Command

The FIND command allows those lines within the basic program which contain a specific text string to be easily located. The format of the command is:

```
FIND <delimited text>[,<line range>]
```

where <delimited text> is the desired text string (32 characters maximum) delimited (preceeded and followed) by any character (except a comma, space, or tab) not contained within the string, and <line range> is a normal line number range specification. If <line range> is omitted, the entire program will be searched.

All lines within the specified range containing the text will be located and displayed on the terminal.

For example:

```
FIND /COS(/, 100-300
```

will find all lines containing "COS(" between lines 100 and 300 inclusive.

1-5. REPLACE Command

The REPLACE command is an extension of the FIND command which not only locates the specified text string, but also replaces it with another string. The format of the command is:

```
REPLACE <delimited text 1><delimited text 2>,<line range>
```

Note that the <line range> is NOT optional on this command.

This command will replace all occurences of <delimited text 1> within the specified range by <delimited text 2>. Each line in which a substitution occurs is displayed in its new form on the console.

If any replacement would cause the line to exceed the maximum size (254 characters) the command is aborted with a "String Too Long" error.

For example:

```
REPLACE /COS(/ \SIN(\, 100-300
```

will replace all occurences of "COS(" between lines 100 and 300 inclusive with "SIN(". Note that the delimiter need not be the same for both strings.

1-6. LOAD Command

The LOAD command has been simplified. The ALOAD command no longer exists. The format of the LOAD command is:

where <file name> is a string value (constant, variable or expression) which contains a standard CP/M file name. If the CP/M file extension (type) is omitted, it defaults to ".BAS".

The specified disk file is located and examined to determine if it contains an ASCII or internal format BASIC program. The appropriate LOAD procedure is then automatically performed. (Note that the internal format of Disk BASIC Version 1.06 programs is significantly different from previous internal formats, and completely incompatible. To transfer from one to another, ASCII format disk files should be used.)

For example:

```
LOAD "PROGRAM"
```

loads a program from the disk file "PROGRAM.BAS".

1-7. SAVE Command

The SAVE command has been simplified. The ASAVE command no longer exists. The format for the SAVE command is:

```
SAVE <file name>[,A]
```

where <file name> is as in the LOAD command. The normal SAVE command (without the "A" option) saves the current program on disk in the specified file in internal format. The use of the "A" option causes the program to be saved in ASCII rather than internal format.

For example:

```
SAVE "PROGRAM",A
```

saves the current program on disk as "PROGRAM.BAS" in ASCII format.

1-8. RESAVE Command

The RESAVE command has been added to simplify the process of working with a single program. The format of the RESAVE command is:

```
RESAVE <file name>[,A]
```

where the arguments are the same as in the SAVE command.

The operation of this command is identical to that of the SAVE command with one exception: the SAVE command gives an error if the specified file already exists on disk; the RESAVE command ERASEs the file if it already exists.

1-9. MERGE Command

The MERGE command replaces the AMERGE command. Its format is:

```
MERGE <file name>
```

where <file name> is as in the LOAD and SAVE commands.

The file specified must be in ASCII format. It is merged, line by line, with the current program.

For example:

```
MERGE "SUB1"
```

merges the program in the disk file "SUB1.BAS" with the current program in memory.

1-10. LOADGO Command

The only change in the LOADGO command is its format. The new format is:

LOADGO <file name>[,<start line>]

where <file name> is as in the LOAD and SAVE commands.

The file specified by <file name> must contain a program in internal format or a "Syntax Error" will occur.

1-11. PRIVACY Statement

To provide for the security of the source for a BASIC program, the PRIVACY statement has been added. The format of the statement is:

PRIVACY <password expression>

where <password expression> is any string value (constant, variable or expression).

When this statement is present in a BASIC program, the source text of the program may only be accessed and modified with the knowledge of the value of the <password expression>. The presence of this statement modifies the syntax of many of the text editing and inputting commands, requiring the prefacing of a password to the command arguments. The commands affected are: LIST, EDIT, DELETE, AUTO, SAVE, RESAVE, COPY, RENUMBER, FIND and REPLACE. In addition, no direct program statement entry or deletion (<line number> <text>) is allowed at all. The password is a string value which must be equal to the declared PRIVACY <password expression> value. It must be followed by a comma if more arguments are to follow. For example, if the source program were:

```
10 A=5
20 B=6
30 D=A*SQR(B)
40 PRINT D
50 PRIVACY "SQUINT"
60 END
```

then the following command would be required to list the entire program:

LIST "SQUINT"

The PRIVACY statement may be anywhere in the program, but MUST be the first statement on the line.

2. Arithmetic and Logical Operators

The set of available arithmetic and logical operators has been expanded. The complete set, in priority order (the order in which they are evaluated) is as follows:

a. Expressions in parentheses "()"

b. ^ (exponentiation)

c. - (unary minus)

d. * and / (multiplication and division)

e. \ (integer division)

f. MOD (modulus)

g. + and - (addition and subtraction)

h. relational operators
   = (equal)
   <> (not equal)
   < (less than)
   > (greater than)
   <= and =< (less than or equal to)
   >= and => (greater than or equal to)

i. NOT (logical bitwise complement)

j. AND (logical bitwise and)

k. OR (logical bitwise or)

l. XOR (logical bitwise exclusive or)

m. EQV (logical bitwise equivalence)

n. IMP (logical bitwise implication)

All operators listed at the same level in the table are evaluated left to right in an expression.

All logical operations convert their operands to sixteen bit integer values prior to the operation. These operands must be in the range 0 to 65,535 or -32,768 to 32,767. An "Illegal Function Call" error will result if the operands are not within this range.

3. Intrinsic Functions

A number of new mathematical and string functions have been added.

3-1. Mathematical Functions

a. LOG10(X) : returns the base ten logrithm of X

b. FIX(X) : returns the truncated integer part of x

c. PI : [no argument] returns the value of pi

d. EE : [no argument] returns the value of e

e. RND : [no argument] returns a random number, same as RND(X) when X>0

f. TIME : [no argument] returns the time in milliseconds since midnight (only on systems with real time clock support, otherwise returns zero)

3-2. String Functions

a. HEX$(X) : returns a string containing the hexedecimal representation of X converted to a sixteen bit integer

b. SPACE$(X) : returns a string containing X spaces (X must be less than 256)

c. STRING$(S$,X) : returns a string containing the string S$ repeated X times (X*LEN(S$) must be less than 256)

d. FIX$(S$,X) : returns a string that is X characters long whose value is S$ either truncated or padded with spaces to the correct length (X must be less than 256)

e. DATE$: [no argument] returns a string whose value is the current date in the form MM/DD/YY (only on systems with real time clock support, otherwise returns a string of eight spaces)

f. TIME$: [no arguments] returns a string whose value is the current time in the form HH:MM:SS (only on systems with real time clock support, otherwise returns a string of eight spaces)

4. Input/Output Operations

The input/output operations in BASIC are significantly changed in this version. Prior to detailed descriptions of each of the various commands, statements, and functions, some basic concepts should be understood.

4-1. Unit Numbers

Because the BASIC now supports multiple I/O devices (console, list, reader, punch, disk), a method is provided to direct a particular I/O operation to a specific device. The mechanism for this is the unit number. Each I/O device, and each active file on the disk, is assigned a unique unit number from 0 to 255. The unit numbers associated with specific devices are fixed as follows:

0 : the console device
1 : the LOAD/SAVE device (normally disk)
2 : the list device
3 : the reader device
4 : the punch device
5-9 : reserved for future expansion
10-255 : the disk device

Note that these devices correspond to the standard CP/M supported devices.

All I/O operations (except LOAD/SAVE) may be directed to any I/O device which is capable of supporting that operation (eg. a PRINT cannot be done to the reader device). The default unit number for all I/O operations (except SAVE/LOAD) is 0 (the console).

The format for specifying a unit number is "#<unit>", where <unit> may be any expression evaluating to a valid unit number. In intrinsic functions which take unit numbers as arguments, the "#" is optional. If the unit number is to be followed by additional arguments, it must be followed by a comma.

For the disk device, any unit number 10 through 255 is valid. The actual association between a unit number and a specific disk file is made by the OPEN command described below.

4-2. Random Addresses

For the disk device, an expanded unit specification is allowed by many of the I/O operations. This specification includes not only the unit number (specifying a particular

disk file), but also an optional random address within that disk file. This random address represents the particular record within the file at which the I/O operation will start (For more information on records, see the OPEN statement). The format of this expanded unit specification is:

#<unit>[@<random address>]

where <random address> is any expression evaluating to a positive integer value less than 4194304.

If specified, the random address is multiplied by the record size to generate a "byte pointer". This byte pointer specifies the particular byte in the disk file at which the I/O operation will start (0 is the first byte in the file). If not specified, the I/O operation will normally proceed from the current byte position (the first byte not processed by the previous I/O operation). The exception to this is files OPENed in the Update mode (see the OPEN statement).

4-3. OPEN Statement

The OPEN statement initializes the I/O device for I/O operation. Each device (and associated unit numbers) has its own specific actions and format for the OPEN statement. The general format of the statement is:

OPEN #<unit>,<mode>{,<file name>[,<record size>]}

where <unit> is as described in Section 4-1, <mode> is a string value (constant, variable or expression) which contains the single character I, O, R or U. The arguments in braces are used for disk units only, and will be described below.

The mode values are as follows:

I Input mode. Only input operations may be done on the unit.

O Output mode. Only output operations may be done on the unit.

R Random mode. Both input and output operations may be done on the unit. Valid only for disk units.

U Update mode. Both input and output operations may be done on the unit. Unless otherwise specified however, each output operation begins at the same byte address in the file as the preceeding input operation, and each

input operation begins at the first unprocessed byte address from the previous I/O operation. Valid only for disk units.

4-3-1. Console (Unit 0)

The console unit is always open. An OPEN to the console unit simply causes a form feed (hex 0C) to be output to the device, and the unit parameters to be reinitialized to the defaults (see Unit Parameters).

4-3-2. List (Unit 2)

The list unit is always open. An OPEN to the list unit (must be mode O) simply causes a form feed (hex 0C) to be output to the list device, and the device parameters to be reinitialized to their default values.

4-3-3. Reader (Unit 3)

The reader unit is always open. An OPEN to the reader unit simply causes the device parameters to be reinitialized to their default values.

4-3-4. Punch (Unit 4)

The punch unit is always open. An OPEN to the punch unit outputs sixteen bytes of leader (hex FF) to the device and reinitializes the device parameters to their default values.

4-3-5. Disk (Unit 10-255)

Disk units are dynamically allocated as requested by OPEN statements. The maximum number of disk units which may be OPENed simultaneously by the program is determined by the units parameter to the CLEAR statement (see the CLEAR statement below).

Each disk unit is associated with a specific disk file. This association is established by the <file name> argument in the OPEN statement. This <file name> is a string value which contains a standard CP/M disk file name. If omitted, the extension (type) is assumed to be ".BAS".

The way in which the association is made is determined by the OPEN <mode> as follows:

I The file is searched for on disk, and if found, the association is made. If not found, an "Input File Not Found" error is given.

O The file is searched for on disk, and if found, an "Duplicate Output File" error is given. If not found, the file is created and the association is made.

R The file is searched for on disk, and if found, the association is made. If not found, the file is created and the association is made.

U The file is searched for on disk, and if found, the association is made. If not found, an "Input File Not Found" error is given.

It should be noted that nothing in the above precludes the same disk file from simultaneously being associated with multiple disk units. In fact, this can on occasion be a usefull technique, as long as care is taken to properly coordinate accesses to the files.

In all modes, the byte address pointer in the file is set to zero as a result of the OPEN, so that all unaddressed I/O operations will begin with the first byte in the file (sequential access).

If a disk unit which is already OPENed is OPENed again without being CLOSEd first, an error does not occur, and the dynamic unit space for that unit is reused. However, the unit is not CLOSEd by this action, and none of the effects of a CLOSE operation will occur.

The final optional argument for a disk unit OPEN is <record size>. This argument must be a sixteen bit integer value between 1 and 32,767. If omitted, it defaults to 1. This value is used to set up all randomly addressed I/O operations. This record size is multiplied by the specified random address to determine the byte address in the file at which the operation will start. Note that a record size of one (the default) will cause the random address and byte address to always be the same. NOTE: This record size only affects those operations in which a random address is specified. It has NO affect on unaddressed (sequential) operations.

4-4. CLOSE Statement

The CLOSE statement release the I/O device being used by a unit and performs device dependent clean-up functions. The format of the CLOSE statement is:

CLOSE [#<unit 1>[,#<unit 2> ...]]

where <unit> is an OPENed unit number. If one or more units are specified in the CLOSE, just those units will be closed. If no units are specified, ALL disk units which are open will closed. The specific actions taken for each type of unit are described below.

4-4-1. Console (Unit 0)

A CLOSE to the console unit only causes the output of a form feed to the console device. No other action takes place, and the unit remains open.

4-4-2. List (Unit 2)

A CLOSE to the list unit only causes the output of a form feed to the list device. No other action takes place, and the unit remains open.

4-4-3. Reader (Unit 3)

A CLOSE to the reader unit has no effect. The unit remains open.

4-4-4. Punch (Unit 4)

A CLOSE to the punch unit causes a CTL-Z (hex 1A) followed by sixteen bytes of leader (hex FF) to be output to the punch device. No other action takes place, and the unit remains open.

4-4-5. Disk (Unit 10-255)

A CLOSE to a disk unit causes different actions depending on the mode in which the unit is open, as follows:

I No specific action takes place.

O A CTL-Z (hex 1A) is written at the current byte address. All memory buffers are updated to disk, and the disk directory is updated.

R All memory buffers are updated to disk, and the disk directory is updated.

U All memory buffers are updated to disk, and the disk directory is updated.

In all modes, the unit is disassociated from the disk file, and the dynamic unit space is released for reuse.

An attempt to CLOSE a unit which is not OPEN will result in a "File Not Open" error.

4-5. Device Parameters

Each I/O unit has associated with it a number of modifiable parameters:

line width : the number of characters output to a line on that unit before a carriage return/line feed is sent automatically

null count and character : the number and value of the characters sent to the device after each carriage return/line feed sequence for timing purposes

quote mode : the character (if any) output to delimit string values when outputting in ASCII mode

prompt character : the character (if any) output to prompt the user that an INPUT statement has been executed

Each of these parameters has a default value for each unit type, and is overidable by the use of the OPTION statement. The format of this statement is:

OPTION [#<unit>,]<option>[,<arg 1>[,<arg 2>]]

where <unit> is as above, <option> is a string value containing either W, N, Q or P. These options take different arguments as follows:

W width : <arg 1> is the width of the line desired (20-253)

N null : <arg 1> is the number of characters to output (0-255) and <arg 2> is the characters decimal value (defaults to 0 if omitted)

Q quote : <arg 1> is the decimal value of the character to be used as the outputted string delimiter (0 means NO delimiter, 34 is a quote mark)

P prompt : <arg 1> is the decimal value of the character to be output as the INPUT statement prompt (0 means no automatic prompt, 63 is a question mark)

Each unit has default parameters as follows:

0 (console) : W[72],N[3,0],Q[0],P[63]

2 (list) : W[72],N[3,0],Q[0]

3 (reader) : not applicable

4 (punch) : W[253],N[0,0],Q[34]

10-255 (disk) : W[253],N[0,0],Q[34]

Note that the OPTION statement replaces the NULL, WIDTH and QUOTE statements of previous versions.

4-6. Dynamic Unit Space

Each disk unit (10-255) requires 181 bytes of memory during the time that it is OPENed. When BASIC is started, the default is to allocate space for no disk units. To change this default allocation, the CLEAR statement is used. The format of the new CLEAR statement is:

CLEAR [<string space>][,<number of units>]

where <string space> is the amount if string area to allocate, and <number of units> is the number of simultaneously OPENed disk units to allocate. If either argument is omitted, the corresponding allocation remains unchanged. The use of the CLEAR statement implicitely results in the disassociation of all OPENed disk units from their corresponding disk files WITHOUT any CLOSE actions being taken.

Note that space is always allocated for the LOAD/SAVE unit (which is effectively a disk unit).

4-7. Data Input and Output

The format of the data input and output commands is unchanged except for the addition of the extended unit specifier option for disk units. The formats are:

INPUT [LINE] [#<unit>[@<addr>],] [<prompt>;] <i/o list>

PRINT [#<unit>[@<addr>],] [USING <format>;] [<i/o list]

READ #<unit>[@<addr>],<i/o list>

WRITE #<unit>[@<addr>],<i/o list>

MAT READ #<unit>[@<addr>],<i/o list>

MAT WRITE #<unit>[@<addr>],<i/o list>

Note the reversal of INPUT LINE from LINE INPUT in previous versions. Also, MSAVE has become MAT WRITE and MLOAD has become MAT READ. The remainder of the statements in each of these cases is unchanged.

It is also important to note that binary I/O (READ and WRITE) operations can only be done to binary devices (not the console [unit 0]).

4-7-1. OUTBYTE Statement

The OUTBYTE statement has been added to facilitate single byte output operations to defined units. The format of the statement is:

OUTBYTE [#<unit>[@<addr>],] <i/o list>

If the <i/o list> element is a numeric value, it must be 0-255, and is output to the specified unit as a single eight-bit byte. If the element is a string, each character of the string is output as a single byte, with no formatting of any sort. The length of the string is not output as it is with a WRITE statement.

4-7-2. SETLOC Statement

The SETLOC statement is provided to set the byte address for a disk unit, independent of any I/O operation. This allows the random address to be determined one place in the program, and all I/O to be done sequentially in another place. The format of the statement is:

SETLOC #<unit 1>@<addr 1> [,#<unit 2>@<addr 2> ...]

where each unit's byte address is set as specified.

4-7-3. ON EOF Statement

The ON EOF statement has been expanded to allow a seperate EOF statement for each unit currently opened. The format is:

ON EOF [#<unit>] [GOTO [<line number>]]

where <line number> is where execution should continue when an End-of-File is encountered.

If the <unit> is specified, than the EOF branch applies only to that unit. If no unit is sepcified, than the EOF branch applies to all units with no EOF branches specified. If the GOTO or the <line number> is omitted, than the EOF branch is cleared.

An EOF branch may be set for the console unit (0), and will be taken whenever a CTL-Z (hex 1A) is received from the console.

The ON EOF may be executed any number of times, and changed as desired.

4-7-4. EOF Statement

In a similar fashion to the above, the EOF statement may cause a software EOF trap on a specific unit. The new format is:

EOF [#<unit>]

If executed, the effect is that of an EOF being encountered on the specified unit.

4-7-5. Intrinsic Functions

A number of new intrinsic functions have been added to increase the flexibility of the I/O system within BASIC. These are as follows:

a. POS(<unit>) : returns the number of characters output to the current line of the specified unit (counted in ASCII mode output only)

b. ERR(<unit>) : returns a logical TRUE (-1) is an I/O error was encountered during the last I/O operation on the specified unit (currently always FALSE [0])

c. EOF(<unit>) : returns a logical TRUE (-1) if an EOF was encountered during the last I/O operation on the specified unit (reset to FALSE [0] at the start of every I/O operation)

d. LOC(<unit>) : returns the byte address of the specified disk unit, the NEXT byte to be sequentially processed (this is always independent of any specified record size)

e. LOF(<unit>) : returns the number of bytes in the current extant of the disk file associated with the specified disk unit

f. BYTEPOLL(<unit>) : return a logical TRUE (-1) if a byte is available from the specified unit (the only time this will be FALSE [0] is for the console unit [0] when no character has

been entered since the last input operation)

g. BYTE(<unit>) : returns the decimal value of the next byte read sequentially from the specified unit

h. BYTE$(<unit>) : returns a string of length one containing the next byte read sequentially from the specified unit

In each of these functions, the <unit> specified must be OPEN.

4-8. Expanded Capabilities for Other I/O

Other commands which perform output have also been enhanced to utilize the new extended unit specification. The formats of these enhanced commands are:

```
LIST          [<password>,]          [#<unit>[@<addr>],]
              [<line number range>]

LVAR [#<unit>[@<addr>]]

TRACE [#<unit>[@<addr>],] <logical value>
```

Note that the previously provided "L" forms of these commands are no longer available (LLIST, LTRACE, LLVAR).

5. Disk File Management

The ERASE and RENAME commands are unchanged, as is the LOOKUP function, with the following formats:

```
ERASE <file name>

RENAME <file name 1>,<file name 2>

LOOKUP(<file name>)
```

In addition, three new file management statements have been added.

5-1. DIR Command

The DIR command allows the display of the disk directory from within BASIC. The format of the command is:

```
DIR [#<unit>[@<addr>],] [<file name>]
```

where <file name> is a string value (constant, variable or expression) containing a valid CP/M disk file name (with ? and * masking if desired). If omitted, the name defaults to "*.BAS", and if the extension is omitted, it defaults to ".BAS". All the files matching the specification are output to the specified unit.

For example, the following command outputs a list of all of the files on disk A to the console:

```
DIR "A:*.*"
```

5-2. PROTECT Statement

The PROTECT command is only applicable to those systems which have individual disk file protection, on all others it does nothing. The format of the command is:

```
PROTECT <file name>,<protection>
```

where <file name> is a string value containing a (possibly masked) CP/M file name, and <protection> is an integer value between 0 and 7 specifying the new protection key.

5-3. RESET Statement

Under CP/M, the changing of a diskette while BASIC (or any program) is running requires updating the operating systems tables. The RESET statement causes that to happen. The format is:

RESET

This command should not be issued with any non-Input mode files OPEN.

6. Program Controlled Console I/O

One of the new I/O functions provided in this version of BASIC is BYTEPOLL. This function determines if a byte of data is ready to be read from an I/O unit. The only unit this really applies to is the console. However, the usefullnes of this function is limited by the fact that BASIC itself constantly tests (and reads) the console input to determine if a CTL-E or other control function has been entered. Hence, under normal conditions, BYTEPOLL will never return a TRUE from the console.

To allow this prgrammed control of console input to work properly, a special statement has been added to the BASIC, the INTERRUPT statement. The format of this statement is:

INTERRUPT <interrupt logical>

The function of this statement is to set the internal console interrupt test to the value of the <interrupt logical>. If that value is TRUE (-1), then BASIC will continue (or resume) testing for CTL-E and other control functions. If it is FALSE (0), then the internal testing will stop. THIS MEANS THAT THE PROGRAM MAY NO LONGER BE STOPPED BY CTL-E. If a program logic error occurs in this mode, and some form of loop takes place, the only method for stopping the program will be resetting the processor.

INTERRUPT is automatically set to TRUE whenver BASIC returns to command mode.

7. Program Execution Control Statements

Two new execution control statements have been added to increase program control over the execution environment. In addition, to avoid a keyword conflict, a new statement has been added.

7-1. Return to Operating System

To leave BASIC and return to the operating system, the BYE command is used. The format of the command is:

```
BYE
```

The execution of the BYE command is TERMINAL. Make sure that all output files are CLOSEd prior to issuing this command, or data may be lost.

This command replaces the EXIT command of previous versions.

7-2. RETURN Statement

The RETURN statement has been enhanced to provide a "non-standard" subroutine return capability. The format of the RETURN command is:

```
RETURN [<line number>]
```

or

```
ON <8-bit value> RETURN <line 1>[,<line 2>,...]
```

where <line number> is the line to RETURN to.

If no line number is specified, the operation of the RETURN command is unchanged. Specifying a line number causes the RETURN to terminate the corresponding GOSUB (as would normally happen), and then continue execution at the specified statement, NOT the statement following the GOSUB.

```
10 GOSUB 40
20 PRINT "LINE 20"
30 STOP
40 PRINT "LINE 40"
50 RETURN 70
60 STOP
70 PRINT "LINE 70"
80 END
```

The RUNning of this program would result in the output:

```
LINE 40
LINE 70
```

The ON ... RETURN format is provided for those cases where the non-standard RETURN is to one of a set of lines depending on some value.

7-3. EXIT Statement

The EXIT statement has been added to allow the correct early termination of a FOR-NEXT loop. The format of the EXIT statement is:

```
EXIT [<line number>][,][<variable name>]
```

where <line number> is the line to EXIT to, and <variable name> is the variable controlling the outermost FOR-NEXT loop to be terminated. Note that the comma is required only if both optional arguments are present.

Due to the fact that the BASIC interpreter allows FOR-NEXT loops to be structured in any fashion (including having the NEXT preceeding the FOR), a mechanism must be provided to specify the point at which the loop is to be considered terminated (as opposed to the normal completion of the loop at the NEXT statement). The EXIT statement provides this capability.

The EXIT statement with no arguments simply terminates the innermost currently active FOR-NEXT loop, leaving the controlling variable with its current value. The addition of a line number causes execution to continue at the specified line after the loop is terminated. This mode of operation is an exact replacement for a GOTO statement in the same context, and should be used whenever it is desired to jump out of a FOR-NEXT loop.

If more than one FOR-NEXT loop is currently active, and it is desired that other than the innermost one be terminated, the variable name for the controlling FOR-NEXT may be specified. In this case, all nested loops within the specified loop are also terminated. Note that the line number may be optionally specified along with a variable.

For example:

```
10 FOR I=1 TO 10
20 IF MX(I)=0 THEN EXIT 60
30 NEXT I
40 PRINT "NO ROOM"
```

```
50 I=0
60 FOR J=1 TO 10
70 FOR I=J TO 10
80 MX(I)=MX(J)
90 IF MX(I)=0 THEN EXIT 140,J
100 NEXT I
110 NEXT J
120 PRINT "DONE"
130 STOP
140 ...
```

illustrates the proper use of the EXIT statement. If a GOTO were used in line 20 rather than an EXIT, a "NEXT without FOR" error would result at line 110 because the second FOR I statement would establish a new active loop at the same level as the previous FOR I, losing the FOR J entirely. With the EXIT statements added, the program works properly.

PM-G12014-0300-01

TDL BASIC VERSION 3

USER'S MANUAL

(Manual Revision 0)

First Printing January 1978

PREFACE

This manual describes the Basic programming language, occupying slightly more than 12k of core, as implemented on a Z-80 based microcomputer.

It describes the unique and powerful commands available in TDL Basic Version 3. Also discussed are the tremendously powerful I/O handling capability, relocatability, ROMability and the large measure of hardware independence.

Great care has been taken to eliminate errors and omissions in this manual. Our technical support staff is available regarding any problems or questions you may encounter. Any effects however, or damages (including consequential) caused by reliance on the material presented, including but not limited to typographical, arithmetic, or listing errors, shall not be the responsibility of T.D.L.

Table of Contents

0.0 INTRODUCTION

0.1 What is a microcomputer?

Microcomputer is a term used to describe any computer built around a microprocessor. Recent advances in integrated circuitry have made it possible to put highly complex functions in a package the size of a domino. The idea of the microprocessor was to make what amounted to a CPU on a single chip. This chip, used as a universal process controller, would be replacing large amounts of complex circuitry in all sorts of equipment.

0.2 What is Basic?

Basic is a programming language most often supplied with small computers. It is very easy to learn, yet offers a great deal of programming flexability, having been originally designed as a beginner's language.

CHAPTER 1

GENERAL INFORMATION

Basic V3 functions as a BASIC Interpreter occupying slightly more than 12k of memory, and provides some of the most advanced software features of any commercially available Basic. If you have already worked with the Zapple 8-K Basic, you will appreciate the new commands that have been added and the extended capabilities of the existing commands.

Basic V3 offers many unique features, including user programmable error handling routines which can process any error occurring in the Basic program without aborting the program; serial input and output of ASCII or binary data files from the Zapple Monitor defined reader and punch devices, and the passing of a variable's address to an assembly language routine which allows routines to return data to the calling program.

All of the above are covered in this manual. This is NOT a "How to write Basic Programs" manual. Many excellent

texts on this subject have been produced. Your local Computer Store can recommend many such texts.

1.0 I/O HANDLING

Oftentimes the lament of the programmer is the lack of source documentation for a Basic Interpreter. This is usually due to the fact that internal I/O routines must be modified to suit the exact configuration of your hardware. The TDL method of I/O handling eliminates this problem in that the source code for the Monitor is provided, and once modified to your hardware configuration, ALL other TDL software automatically interfaces to your system.

Basic V3 has this feature of hardware independence. All of its I/O drivers are contained in the Monitor, so interfacing it to your hardware is simple.

To get the most out of BASIC, we highly recommend getting the 2K ZAPPLE MONITOR.

1.1 LOADING BASIC WITH ZAPPLE

Loading of Basic V3 is very straight-forward. It is loaded using the "R" command of either the Zap or Zapple Monitors. It is provided on paper tape in TDL's relocatable hex file format. It occupies slightly more than 12K of core.

Basic V3 has been assembled on TDL's relocating macro-assembler. Because of this, Zapple Basic is completely relocatable. It is not necessary to load and run this program at one address only. Within limits, which will be mentioned here, it may be loaded and run at any convenient address by the user.

The procedure for loading the program is very simple. Place the tape in the reader device on the nulls between the serial number and the start of the data. Type on the console: "R,(x)"(cr): and start the reader.

Example: R,300 (cr)

will load Basic V3 at address 300H. For the exact details on the operation of the "R" command, see either the Zap or Zapple Monitor Manuals.

After loading, Basic V3 will NOT sign on. You must begin execution at the address given in the relocation parameter above by typing: G300 (cr). Basic will then ask: "Highest Memory?" asking the user to type in decimal the upper limit Basic will be allowed to use. A carriage return (cr) will assign all available memory to Basic save for a small amount at the top which is reserved for use by the monitor.

The limits on its practical relocatability are governed by two factors; The buffer storage area required, and the address at which the monitor will be located.

The first factor is that, Basic V3 requires a buffer space of approximately 512 bytes. Regardless of the loading address of Basic, this 512 byte buffer resides from address 100H to 2FFH. Thus, the minimum loading address for Basic V3 is 300H. From this it should be evident that this Basic at no time uses any memory below address 100H, as is the case of 8-K Basic.

THE MINIMUM LOADING ADDRESS FOR BASIC V3 IS 300H.

As to the second factor, the Monitors are also relocatable, but we do recommend that they be placed up near the top of memory at F000 (up "out of the way"). Thus the 2K monitor would reside from F000 to F7FF (hex) allowing F800 to FFFF for monitor extension routines. (Such as a VDM driver, Tarbell driver. Etc.). Thus, since the Basic V3 occupies slightly more than 12K of core, the maximum practical loading address is B000 (hex).

Basic will normally use the location of the initialization routine as an input buffer, thus once Basic signs on, and you type in a program you must reenter Basic at the recovery point (loading address +3). If, during the sign on sequence, Basic gets the highest memory address that is below the beginning of Basic then Basic assumes that it is running in ROM and does not attempt to modify itself.

1.2 ZAPPLE BASIC VERSION 3 JUMP TABLE

All I/O handling for Zapple Basic is done through either the ZAP (1K) or ZAPPLE (2K) monitors. The I/O interfacing is done in the beginning of the Basic V3 Program.

Zapple Basic has in addition, a recovery address, from which recovery of the program can often be made following a

"blow-up".

Zapple Basic V3 also has a "USR" command which allows one's own assembly language routines to be called as part of a Basic program. A jump vector is provided allowing the user great latitude in this application.

On Page 5, the source code of the first part of Zapple Basic V3 is presented. It contains all of the I/O vectors which are necessary for complete user versatility. Note that as part of the code, addresses marked with an apostrophe (') are those addresses which are relocatable. Those without an apostrophe (') are considered absolute, in that they are vectoring to addresses outside of Basic, where they expect to find specific I/O routines.

The specific I/O routines in question are those of the monitor. Although both the Monitor and Zapple Basic are relocatable, we recommend placing the monitor (either the Zap or Zapple) as high as possible - usually F000H. Thus Basic expects to find the monitor at that address. The source code of these "jumps to the monitor" are presented so that in the event that you do not wish to, or are not able to have the monitor reside at this address, you may make the necessary, although simple modifications to the program.

Note that the source code below is in TDL's relocating assembler format, in that address information is presented in the "High byte first, low byte second" format. For example, at address 000C' there is a jump to F006 (hex). With some assemblers this might be construed to be a jump to address 06F0. Also note that any modifications you might make when using the monitor at an address other than F000 would entail changing only the high byte of the stated jump address.

```
0000' C3 XXXX'    BASIC:   JMP   INIT    ;"INITIALIZE"
                                           ENTRY POINT
0003' C3 XXXX'    REST:    JMP   RECOVER;RECOVERY ENTRY
                                           POINT
0006' C3 XXXX'    USR:     JMP   ERROR   ;USER DEF.
0009' C3 F003     CI:      JMP   CIN     ;CONSOLE INPUT
000C' C3 F006     RI:      JMP   RIV     ;READER INPUT
000F' C3 F009     CO:      JMP   CON     ;CONSOLE OUTPUT
0012' C3 F00C     PO:      JMP   WRTV    ;PUNCH OUTPUT
0015' C3 F00F     LO:      JMP   LISTX   ;LIST OUTPUT
0018' C3 F012     CSTS:    JMP   CSTSX   ;CONSOLE
                                           STATUS CHECK
001B' C3 F015     IOCHK:   JMP   IOCHX   ;I/O CONFIG.
                                           CHECK
001E' C3 F018     IOSET:   JMP   IOSTX   ;I/O MODIFCTN.
0021' C3 F01B     MEMSIZ:  JMP   MEMCK   ;MEMORY SIZE CK
0024' C3 F01E     TRAP:    JMP   TRAPX   ;BREAKPOINT ENTRY
```

"X"s are inserted into some jump notations above because the values may change in future versions of Basic, and thus could cause confusion. These addresses are modified in the course of various applications, and all that is needed is the recognition that they lie at the starting address of basic (where it was loaded), plus the address at the beginning of the line.

Specifics on making use of the USR command portion of the above are covered in the section of the manual which deals with the USR command.

1.3 ERROR MESSAGES

The following is a list of error messages returned by Basic when the particular error has been detected. They are given here without explanation. Within the context of a Basic program they indicate clearly what is amiss, and are of great use in program debugging.

1	NEXT W/O FOR
2	SYNTAX ERROR
3	RETURN W/O GOSUB
4	OUT OF DATA
5	ILLEGAL FUNCTION
6	ARITHMETIC OVERFLOW
7	OUT OF MEMORY
8	UNDEFINED STATEMENT
9	SUBSCRIPT OUT OF RANGE
10	RE-DIMENSIONED ARRAY
11	CAN'T /0
12	ILLEGAL DIRECT
13	TYPE MIS-MATCH
14	NO STRING SPACE
15	STRING TOO LONG
16	TOO COMPLEX
17	CAN'T CONTINUE
18	UNDEFINED USER CALL
19	FILE NOT FOUND
20	ILLEGAL EOF
21	FILES DIFFERENT
22	RECOVERED
23	FNRETURN WITHOUT FUNCTION CALL
24	MISSING STATEMENT NUMBER
25	RECORD TOO LARGE
26	UNDEFINED MATRIX
27	INVALID UNIT NUMBER
28	RESUME W/O ERROR

CHAPTER 2

2.0 BASIC VERSION 3 COMMAND SET

COMMAND	GROUP	PURPOSE
ABS	8	Absolute Value Function
ALOAD	11	Loading of ASCII Source Programs
ALOAD*	11	" " " "
AMERGE	11	" " " "
AMERGE*	11	" " " "
AND	6	Logical AND operator
ASAVE	11	Allows punching of ASCII text.
ASC	9	Convert character to numeric value function
ATN	7	Arctangent function
AUTO	1	Provides automatic generation of line numbers while a Basic program is being entered.
CALL	6	Invokes assembly language subroutines
CHR$	9	Convert numeric value to character function
CLEAR	1	Delete all variables and set string space
CONT	1	Continue program execution from program breakpoint
COPY	5	Provides method of moving, or duplicating one section of a Basic program into another part of the program
COS	7	Cosine function
DATA	5	Defines constants
DEF	8	Defines User functions
DELETE	1	Deletes range of line numbers
DIM	5	Reserves storage for matrices
EDIT	2	Invokes the line editor
ELSE	6	What to do if relational is not true
END	10	End of program; return to command mode
EOF	6	Causes a software controlled initiation of the end of file processing.
ERL	14	Function to return the line number at which an error occurs.
ERR	14	Function to return the line number of error which last occurred.
ERROR	14	Allows the use of software generated errors in conjunction with the error trapping capability.
EXCHANGE	5	Exchanges the values of two variables without the use of a third variable.

COMMAND	GROUP	PURPOSE
EXP	8	Function to return "e" raised to a power
FN	8	Class of user defined functions
FNEND	8	Multiline functions
FNFAC	8	" "
FNRETURN	8	" "
FOR	6	Sets up a loop
FRE	8	Function to determine amount of unused memory.
GOSUB	6	Invokes a subroutine
GOTO	6	Transfer control to another part of the program
IF	6	Relational test
INP	5	Input directly from an I/O port
INPUT	5	Input data from the keyboard
INSTR	9	Searches one string for a specified substring
INT	8	Function returns the integer portion of a number
KILL	1	Allows unneeded matrix space to be returned to the system
LEFT$	9	Returns the left portion of a string
LEN	9	Returns the length of a string
LET	5	Logical Assignment
LINE INPUT	5	Provides increased flexibility in handling console input.
LIST	3	Lists the program on the console
LLIST	4	Lists the program on the list device
LLVAR	4	Lists the program variables on the list device
LNULL	4	Sets the nulls for the printer
LOAD	1	Loads a program from the reader
LOADGO	1	Allows one Basic program to load and transfer control to another.
LOG	8	Returns the natural logarithm of a number
LPOS	4	Function to return the current position of the list device
LPRINT	4	Outputs on the list device
LVAR	3	Prints the variables on the console
LWIDTH	4	Sets the width of the list device
MID$	9	Returns the middle of a string
MLOAD	5	High speed input of an entire numeric matrix at one time, in binary.
MSAVE	5	High speed output of an entire numeric matrix at one time, in binary.
NEW	1	Clears all program statements and variables
NEXT	6	Returns to the beginning of a loop
NOT	6	Logical "NOT" operator

COMMAND	GROUP	PURPOSE
NULL	3	Sets the nulls for the console
ON	6	Indexed transfer of control
ON EOF GOTO	6	Allows the user to detect and process the end of file condition.
ON ERROR GOTO	14	Specifies a user error handling procedure.
OR	6	Logical "OR" operator
OUT	5	Output directly to an I/O port
PEEK	5	Function to return data from a memory location
POKE	5	Insert data into a memory location
POS	3	Function to return the correct print head position of the console
PRECISION	1	Allows the specification of a default print-out precision of less than 11 digits
PRINT	3	Directs output to console
PRINT USING	3	String and numeric specifications
QUOTE	5	Provides output which is directly readable by an INPUT statement.
RANDOMIZE	8	Changes the seed used by the pseudo-random number generator
READ	5	Move data from a DATA statement to a variable
REM	10	Remarks
RENUMBER	1	Renumber the program and change line number references
RESTORE	5	Returns pointer to the beginning of the data statements
RESUME	14	Routine to return control to regelar program execution after error processing.
RESUME NEXT	14	Starts execution at statement following the one causing the error.
RETURN	6	Return control back from a subroutine
RIGHT$	9	Function to return the right portion of a string
RND	8	Function to return a pseudo-random number
RUN	1	Clear variables and start execution of program
SAVE	1	Dump a copy of the program to the currently assigned punch device
SGN	8	Function to return the sign of a variable
SIN	7	Sine function

COMMAND	GROUP	PURPOSE
SPC	3	Used in PRINT statement to print spaces
SQR	8	Function to return the square root of a number
STEP	6	Used in FOR statement for increment of loop control
STOP	10	Used to terminate program execution
STR$	9	Function to convert a value to a character string
SWITCH	3	Used to change the console assignment
TAB	3	Used in a PRINT statement to tab to a position
TAN	7	Tangent function
THEN	6	What to do if relational IF is true
TO	6	Used in FOR statement to specify limit
TRACE	3	Used to turn ON/OFF line number trace
USR	10	May be patched to user provided routine
VAL	9	Function to return the numeric value of a string expression
VARADR	6	Function to allow the actual address that a particular variable resides at in memory be obtained.
WAIT	5	Used to loop on a status port
WIDTH	3	Set the width of the console
WRITE	5	Binary output statement to write to output device.
HEXADECIMAL CONSTANT	9	Constant used directly in the Basic program
?	13	Same as PRINT
UP-ARROW	13	Exponentiation operator
-	13	Subtraction operator
*	13	Multiplication operator
/	13	Division operator
+	13	Addition operator
<	13	Less than operator
>	13	Greater than operator
=	13	Equals operator

COMMAND	GROUP	PURPOSE
CONTROL U	12	Delete input line
CONTROL C	12	Abort execution of program
CONTROL X	12	Return to monitor
CONTROL O	12	Suppress console output
CONTROL R	12	Allows more input to be entered
CONTROL T	12	Prints line number of line currently being executed
CONTROL S	12	Temporarily stop execution
CONTROL Q	12	Restart the program
RUBOUT	12	Delete previous character
, (comma)	12	Move to next TAB position or delimiter
;	12	Don't move
:	12	Used for multiple statements per line

NOTE: The control characters and operators listed at the end of the COMMAND SET LIST, while not exactly being BASIC commands, are included here for your reference and convenience.

CHAPTER 3

3.0 DETAILED DESCRIPTIONS OF COMMANDS/FUNCTIONS

All numeric calculations are carried out to twelve significant digits, and rounded to eleven digits. This includes all intrinsic functions (eg. TAN, SIN, etc.).

Each statement line may be up to 255 characters long. The line may be formatted for additional readability with both tab and space characters, and may be spread over multiple physical lines to emphasize program structure by use of the line-feed character. A tab character will cause the line, when listed, to be spaced to the next multiple of eight column. A line feed will list as a carriage-return/line-feed. Tabs, extra spaces, and line feeds are all ignored during the processing of the statements.

For example:

100<tab>IF I=23 THEN<line feed><tab>I=0:<line feed><tab>J=1

will list as:

```
100     IF I=23 THEN
        I=0:
        J=1
```

The statements would be executed as if they had been entered all on the same line.

An error does not cause a loss of program context. Even if the error causes a program abort, all currently active function calls, FOR loops, GOSUBS and error traps are preserved. It is then possible to examine or modify variable values or examine program statements, and to restart the program through a direct mode GOTO command. Any modification to the program itself will result in the loss of all variables and the program context. (See GROUP 14 ERROR HANDLING).

File number is used where <file number> is 0=console, 1=punch, and 2=list device. I/O list is used where <I/O list> is a list of variables specifying where to get the data from/or where to put the data.

3.1 GROUP 1 GENERAL PURPOSE UTILITY COMMANDS

AUTO

The AUTO command provides for automatic generation of line numbers while a BASIC program is being entered. The format of the command is:

AUTO [<starting number>] [,<increment>]

If <starting number>] is omitted, 10 is assumed, and if <increment> is omitted, 10 is assumed. To terminate the automatic numbering, an empty line (just a carriage return) should be entered. If a line is entered whose generated number is the same as an existing line, the new line replaces the old one.

CLEAR

Deletes all variables in storage. The command CLEAR followed by an argument, such as, "CLEAR 400" will set the string space to that value.

The CLEAR command may be placed in a program,

For example:

15 CLEAR 250

to set the string space to the exact amount needed by that program. If the argument is omitted, the string space is not changed.

CONTINUE

If your program is stopped by typing a control C or by executing a stop statement, then you may resume execution of your program by typing CONTINUE OR CONT. Between the stopping and restarting of the program you may display the value of the variables, (See PRINT and LVAR) or change the value of the variables, (see LET). However, you may not modify the program, or continue after an error.

DELETE

Deletes a range of line numbers. The DELETE command is followed by two line numbers separated by a dash "-". For example, DELETE 115-135 would delete from your program the line numbers 115 up to and including 135.

DELETE 25 would delete only line 25.

KILL

The KILL command allows unneeded matrix space to be returned to the system. The format of the command is:

KILL <matrix>1 [,...,<matrix n>]

Each name specified must be a matrix id with no subscripts following. Reference to a matrix which has not been defined is not an error. If the matrix has been defined, all the space being used by the matrix for variable storage and for matrix information will be returned to the system, and the matrix will be undefined.

LOAD

Loads a program from the reader device. The LOAD command is followed by a string expression which evaluates to a single character program id. A NEW operation is performed (see NEW), then the reader device is searched for a program under that name, if found, the program is then loaded. Example: LOAD "P"

A "bell" will sound on the console when the file starts to load. Additionally, a saved file may be verified by reloading the file using the following format:

LOAD?"P"

If an error occurs a message will be generated, otherwise, you will return to the command mode.

Example: A$="X"

```
LOAD A$
  or
LOAD "X"
```

LOADGO

The LOADGO command is used to allow one Basic program to load and transfer control to

another. The format of the command is:

LOADGO <file id>[,<starting line>]

The command functions similarly to the LOAD command. When executed, it searches the reader device for an internal format program whose id is a string expression which evaluates to a single character program id <file id>. If found, the program is loaded, and control is transferred to it. The new program either begins at its first statement, or if the optional argument was given to the LOADGO command, at the specified line number.

Example:

LOADGO "P",1000

It is important to note that the LOADGO command clears the program area before initiating the file search. Also, all data values are cleared prior to the loading of the new program.

NEW

This command deletes all program statements and any stored variables. The slate is wiped clean, so to speak.

PRECISION

Because the added precision of this version of BASIC sometimes can produce long fractional results, the PRECISION command allows the specification of a default print-out precision of less than 11 digits. The format of the command is:

PRECISION [<digits>]

If <digits> is zero or omitted, the normal precision of 11 digits is restored. Otherwise, the precision is set as specified. This precision affects all numeric output not specified by a PRINT USING format. The internal calculations are still performed to the same accuracy (11 digits). The number is rounded to the desired precision before display.

RENUMBER

The RENUMBER command causes the lines of a program to be renumbered and all the internal line number references such as 123 GOTO 547 to be properly adjusted. This command has three parameters separated by a comma. The first parameter specifies the starting point for the line numbers, the second parameter specifies the increment between numbers. If either parameter is omitted it defaults to 10. The third optional argument specifies the current line number of the first line to be renumbered. The use of this option allows a "hole" to be inserted into a program to allow for the insertion of additional lines.

Format:

RENUMBER [<new number>][,<increment>]
[,<start line>]

Example:
```
RENUMBER
RENUMBER 10
RENUMBER ,10
RENUMBER 10,10
```

All start at 10 and increment by 10.

```
RENUMBER 40,10,60
```

Starting at line 60, change line 60 to 40 and renumber by 10.

RUN

RUN clears all variables and starts the execution of the program starting with the first program statement. RUN followed by a line number will clear all variables and start execution at that line number.

Example:
```
RUN 105
```

SAVE

This command causes a copy of the program to be output to the punch device using Basic's internal compressed format. The SAVE command has one parameter, a string expression which evaluates to a single character program id. The program name is used in reloading the program with the LOAD command. Also the comment string in the SAVE command may now be an arbitrary string expression, rather than

just a string constant.

Example: SAVE "P","<message>"

saves the program under the name "P".

Note: Only the 26 uppercase characters are valid as program names. All other characters will generate a SYNTAX ERROR message.

3.2 GROUP 2 THE EDIT COMMAND

EDIT The EDIT command followed by a line number invokes the line editor to process that line. The editor mofes a copy of the line to be edited into its edit buffer. At the end of the editing process, the user has the option of replacing the line in the program with the contents of the edit buffer, or throwing away the changes (say that you decide that you really don't want to make those changes).

Example: EDIT 55

The editor will then print the line number and then wait for single letter commands. ALL commands are NOT ECHOED. Illegal commands will echo as a bell. Some commands may be preceeded by a numerical instruction to repeat itself. These are shown by a lower case "n" and may range from 1 to 255.

EDIT COMMAND/ FUNCTION

A Reload the Edit Buffer from the program line. This is used after making a mistake.

nD DELETE "n" characters

E END EDIT - don't print line and replace program line with the contents of the edit buffer.

nFx FIND the "n"th character X in the edit buffer and stop with the pointer just before the character.

H DELETE everything to the right of the pointer and go to the insert mode.

I INSERT all following characters from the keyboard, STOP INSERTING on a carriage-return or Escape.

nKx KILL or DELETE characters from where the pointer is now to the "n"th character X, but don't delete that character.

L Print the line and return to the beginning of the line.

Q	QUIT. Leave the edit mode - without replacing the program line.
nR	REPLACE the "n" following characters with characters from the keyboard.
X	MOVE the pointer to the end of the line, and go to the insert mode.
SPACE	MOVE the pointer to the right.
RUBOUT	MOVE the pointer to the left.
CARRIAGE RETURN	End Editing, print the line, replace the program line. This may also be used to terminate the insert mode.
ESCAPE	END insert mode or cancel pending commands.

In the following edit examples an exclamation mark (!) is used to show the position of the console print head or cursor. This line:

```
55 PRINT A,B;"DOLLARS"
```

will be used in all the examples.

USER TYPES: MACHINE RESPONDS:

```
EDIT 55      55!
```

TO LIST OUT THE LINE: (command not echoed!

```
L            55 PRINT A,B;"DOLLARS"
             55!
```

TO MOVE THE POINTER FORWARD:

```
             (space)  55 P!
             (space)  55 PR!
             10(space)55 PRINT A,B;"D!
```

TO MOVE THE POINTER BACKWARD, THE SYSTEM ECHOS CHARACTERS THAT ARE PASSED BY THE POINTER AS IT IS GOING BACKWARDS.

```
(rubout)   55 PRINT A,B;"DD!
2(rubout)  55 PRINT A,B;"DD";!
L          55 PRINT A,B;"DD";;"DOLLARS"
           55 !
```

Although the example of what happens when using the rubout command may be confusing when shown as above, in actual use, the response of the machine when you type the rubout command is quite easy to get used to.

TO DELETE A CHARACTER:

```
D          55 \P\!

L          55 \P\RINT A,B;"DOLLARS"
           55 !

L          55 RINT A,B;"DOLLARS"
           55 !

(space)    55 R!

2D         55 R\IN\T A,B;"DOLLARS"

L          55 R\IN\T A,B;"DOLLARS"
           55 !

L          55 RT A,B;"DOLLARS"
           55 !
```

TO RECOVER FROM A MISTAKE:

At this point we decide we didn't really want to change the word "PRINT", so we can reload the edit buffer with the "A" command.

```
A          55 !

L          55 PRINT A,B;"DOLLARS"
           55 !
```

The INSERT command inserts characters into the line at the present position of the pointer. The INSERT command is terminated by either an escape character or a carriage return.

```
L              55 PRINT A,B;"DOLLARS"
               55 !

18(spaces)     55 PRINT A,B;"DOLLARS!

IXX(escape)    55 PRINT A,B;"DOLLARSXX!

L              55 PRINT A,B;"DOLLARSXX"
               55 !

L              55 PRINT A,B;"DOLLARSXX"
               55 !
```

The "X" command moves the pointer to the end of the line and goes into the insert mode.

```
XYY(escape)    55 PRINT A,B;"DOLLARSXX"YY!

L              55 PRINT A,B;"DOLLARSXX"YY
               55 !

L              55 PRINT A,B;"DOLLARSXX"YY
               55 !
```

The "H" command deletes all the line to the right of the pointer and goes to the insert mode.

```
11(spaces)     55 PRINT A,B;"!

HCENTS"(esc)   55 PRINT A,B;"CENTS"!

L              55 !

L              55 PRINT A,B;"CENTS"
               55 !
```

3.3 GROUP 3 COMMANDS INVOLVING THE CONSOLE

LIST

Causes the program to be typed on the console device. As a logical extension of the file number concept, the output control command LIST will accept an optional file number. List may have one or two parameters separated by a dash.

Format:

LIST #<file number>,[line number-line number]

An omitted file number will default to the console (0).

Example: LIST #2,20-30

will print lines 20 thru 30 on the list device. LIST 20 would only print line 20. LIST 20- lists from 20 to End.

LVAR

Causes the variable storage area to be typed on the console. As a logical extension of the file number concept, the output control command LVAR will accept an optional file number as follows:

LVAR #<file number>

An omitted file number will default to the console (0).

NULL

Sets the number of nulls to output to the console after a carriage return line feed sequence. This may be used to give additional time after a carriage return for terminals which require additional time to return the print head.

The NULL command may be followed by upto 3 parameters separated by a comma. The first parameter is an optional file number. The second specifies the number of nulls to send after the carriage-return/line-feed sequence and the third, as a decimal number, specifies

what character is to be used. (initializes to zero or ASCII null).

Format:

NULL [#<file number>,] <number> [,<character>]

An omitted file number will default to the console (0).

Example: NULL 3,255

sends 3 rubout characters after a carriage-return/line-feed to be output to the console.

POS

This function is used to return the present position of the print head or cursor of the console device. The first position is considered to be zero (0). The function POS accepts an optional file number.

Format:

POS(<file number>)

An omitted file number will default to the console (0).

Example: 40 A=POS(B)

where B=0 for the console or 1 for the punch or 2 for the list device. The example above would take the present position of the print head as a number from 0 to 131 and place it in variable A.

NOTE: LPOS will take a dummy argument since it already implies the line printer as the list device.

PRINT

The PRINT command is used to direct printed output to the console device. An optional <file number> parameter can be used to direct output to any of the three devices (0=console, 1=punch, 2=list device). The default is the console. The PRINT command is followed by a list of variables,constants, or literals to be printed, separated by commas or semicolons.

These are the variables in the <I/O list>.

Output in ASCII mode uses the PRINT and PRINT USING commands in the following format:

PRINT [#<file number>,] [USING <format>,] [#<file number>,] [<I/O list>]

where <file number> is again an expression evaluating to a valid file number. It can be located at either point indicated when the USING option is used. Valid file numbers for ASCII output are 0 for the console, 1 for the punch, and 2 for the list device (Note that file 2 overlaps the Lxxxx commands. The Lxxxx commands will be removed in a future version of Basic, so the new format is to be perferred). <I/O list> is a list of variables specifying where to get the data from. The PRINT command outputs exactly the same format to any of the three devices.

Example: 40 PRINT #2,123,A,"IS THE ANSWER"

If the separators are commas, then the items are printed in columns spaced every 14 positions across the list device.

If the separators are semicolons, the items are printed with 2 spaces in between. Additionally, if the last item in the print list is followed by a semicolon, then unless executing the command would overprint the last print position on the console, BASIC will not carriage return but would stack successive PRINT commands across the console on the same line.

```
Example:  10 PRINT A,B;
          20 PRINT C
```

would print A,B and C across the same line on the console. There would be 14 spaces between the beginnings of A and B and 2 spaces between the end of B and the beginning of C.

PRINT USING The PRINT USING statement has two formats available:

PRINT USING [#<file number>,] <line number>[;<I/O list>]

PRINT USING [#<file number>,] <string value>[;<I/O list>]

In the first format, the <line number> refers to a "format line" which contains the format string. This line must start with an exclamation point (!) and be followed by the format specification. In the second format, the format specification is taken from the string value specified.

For example:

```
100  A = 5
110  PRINT USING 120;A
120  !##.##
130  PRINT USING "##.##";A
```

Both lines 110 and 130 would print a space followed by the characters 5.00

The format specification consists of any valid combination of the following field specifiers:

FORMAT SPECIFICATIONS

NUMERIC

\# Numeric fields are specified by use of the #. Each # in a field represents one digit position. The number will be right justified in the field, with leading spaces added to fill the field.

. Decimal point alignment is specified by the use of a decimal point. The number will be rounded to fit the specification. A digit before the decimal point will always be filled (with a zero if necessary). For example:

```
###.## 23.456 =>  23.46
###.### -24.5 => -24.500
###.## .12345 =>   0.12
```

\+ A plus sign may be used either at the start or the end of a numeric specification. It will force the + sign to be printed at that end of the field if the value is positive (normally a space would be printed). A - sign will be printed in that position if the number is negative.

- A minus sign may be used at the end of the numeric specification to indicate that - sign for a negative number should be printed at the end, as opposed to the start, of the number. If the number is positive, a space is printed.

** Two (or more) asterisks at the start of a numeric field indicate asterisk "fill" of the field. Each asterisk indicates one digit position, and all empty digit positions prior to the decimal point will be filled with asterisks instead of spaces.

$$ Two (or more) dollar signs at the start of a numeric field indicate a "floating" dollar sign. This means that the dollar sign will be placed immediately adjacent to the first non-zero digit in the number. Each dollar sign indicates one digit position, but one of these positions is occupied by the dollar sign itself.

**$ This indicates the combination of the two above features.

, A comma anywhere to the left of the decimal point indicates that commas are to be inserted every three digits. Each comma also indicates one digit position in the specification.

^^^^ Four up-arrows at the end of a numeric specification indicate that the number is always to be printed in exponential notation. Decimal point alignment is allowed, but the significant digits are left justified and the exponent is printed accordingly (E+nn or E-nn).

If any numeric value will not fit in the field specified, it will still be printed in its entirety, but will be preceeded by a percent sign (%).

STRING

String fields are specified by use of the apostrophe ('). A single apostrophe indicates a single character string field. Multiple character string fields are specified by following the apostrophe with one (or more) of the letters C, R, L, or E. The size of the field is equal to one plus the number of letters following. The letters have the following meanings.

L The string value is left justified in the specified field. If the value is longer than the field, extra characters are lost on the right.

R The string value is right justified in the specified field. If the value is longer than the field, extra characters are lost on the left.

C The string value is centered in the specified field. If the value is longer than the field, extra characters are lost on the right.

E The string value is left justified in the specified field. If the value is longer than the field, the field is Extended to allow the entire string value to be printed with no lost characters.

Note that the format string will be reused until all values in the output list have been printed. The printing of the format string will terminate when a field specification is encountered and the output list is empty. Any character in the format specification which is not part of a numeric or a string specification will be printed literally on the output device at the specified position in the string.

SPC

This is a function-like command that is used to print a number of spaces on the console. It is only used in a PRINT or LPRINT statement and is called function-like because it looks like a function but CANNOT be used in a LET statement.

Example: 35 PRINT A;SPC(5);B

may be used to place an additional 5 spaces between A and B over and above the two that would normally be printed due to the semi-colon.

SWITCH

This command is used to change the console assignment. SWITCH used with no variable will always switch between the teleprinter and the user console. SWITCH with an argument of 0-3 (zero to three) will assign the console to that value. I.E.

0=TTY
1=CRT
2=BATCH MODE
3=USER DEFINED

A value greater than 3 will generate an error message. These are further discussed in the ZAPPLE Monitor Manual.

TAB

TAB is a function-like command that is used only with a PRINT or LPRINT statement and is used to tab directly to a particular position. If the printhead or cursor is on or after the specified TAB position then BASIC will ignore the TAB command.

Example: 25 PRINT A;TAB(25);B

TRACE

This is a one parameter command that turns on or off the TRACE function. If the TRACE is on, then Basic will print the line numbers of the statements executed enclosed in angle brackets, e.g. <25>. The parameter may be an expression and if that expression is evaluated to be non-zero, then the TRACE is turned on. If the expression is evaluated to be zero, then the TRACE is turned off. (NOTE: The TRACE and LTRACE are completely independent functions and may be separately manipulated at will.)

Example: 25 TRACE A-B

If A-B is equal to zero then the TRACE is turned off, if equal to non-zero then TRACE is turned on.

WIDTH

Basic keeps track of the number of characters and spaces printed on the console and will generate an automatic carriage-return/line-feed to prevent overprinting at the end of a line. The WIDTH command may be used to change the sign-on default of 72 spaces. WIDTH accepts an optional file number. (0=console, 1=punch, 2=list device)

Format:

WIDTH [#<file number>,] <width>

Example: WIDTH 80

will cause an automatic carriage-return/ line-feed sequence after 80 characters. The minimum value is 15, and the maximum value is 255.

3.4 GROUP 4 COMMANDS INVOLVING THE LINE PRINTER

Most of the commands of GROUP 3, which affect the console, have a counterpart in GROUP 4, which affects the line printer. The general rule is to add the letter L in front - thus PRINT becomes LPRINT, and LVAR becomes LLVAR, etc. This section simply lists the commands and directs your attention back to GROUP 3 for information on how the commands function otherwise. These commands will eventually be removed in a future version of Basic. The use of the <file number> which is an expression evaluating to a valid file number (i.e. 0 for the console, 1 for the punch and 2 for the list device) will take precedence over the Lxxxx commands.

LLIST	see LIST
LLVAR	see LVAR
LNULL	see NULL
LPRINT	see PRINT
LPRINT USING	see PRINT USING
LTRACE	see TRACE
LWIDTH	see WIDTH
LPOS	see POS
SPC	use in LPRINT statement
TAB	use in LPRINT statement

3.5 GROUP 5 COMMANDS AND FUNCTIONS THAT INVOLVE THE MOVEMENT OF DATA FROM ONE PLACE TO ANOTHER.

LET(=)

This is the assignment command. It causes the evaluation of an expression on the right side of the equals sign (=) and the assignment of the resultant value to the variable on the left side of the equals sign.

Example: 10 LET A=B+2

would cause BASIC to get variable B, add 2 to it, and place the result in A. In TDL Basic, the command LET is optional. For example, the previous statement could also be written as:

10 A=B+2

DIM

Reserves storage for matrices. The storage area is first assumed to be zero. Matrices may have from one to 255 dimensions, but is limited by the available remaining workspace (memory).

```
257  DIM A(72),B(4)
258  DIM C(72,66)
259  DIM D(J)
```

Matrices may also be dimensioned during the execution of the program after the storage space is calculated, however, remember that the DIM command zeros the storage area, and a previously dimensioned array may not be re-dimensioned.

DATA

Specifies constants that may be retrieved by the READ statement.

Example: 10 DATA 5,4,3,2,1.5

READ

Retrieves the constants that are specified in the DATA statement. Binary I/O is performed by the use of the READ and WRITE commands. Binary data is written in internal

format, and is only useful to other Basic programs. The format of the input command is:

READ #<file number> [,<I/O list>]

This command performs two functions, depending on whether an I/O list is present or not. If no I/O list is present, the command causes the input data stream to be searched until a binary data header is found (seven 0FFH followed by one 00H). The input is positioned at the first byte following the header, and the command is done. If an I/O list is present, then sequential bytes are read from the input device until the I/O list is satisfied. It is important to note that Basic does no checking on the validity of the incoming data. It is the users responsibility to read the data in a way compatible with that in which it was written.

Example: 20 READ #1,A

The first time line 20 is executed the value 5 from the previous example will be placed into variable A. If at some later time statement 20 is executed again, or another READ statement is executed, then the value 4 would be retrieved etc. DATA statements are considered to be chained together and appear to be one big data statement. If at any time all the data has been read and another READ statement is executed then the program is terminated and the message "OUT OF DATA @ LINE (N)" is printed.

RESTORE

This command restores the internal pointer back to the beginning of the data so that it may be read again. It takes an optional line number as an argument. If specified, the DATA read pointer will be set to the specified line instead of the start of the program. The format of the command is:

RESTORE [<line number>]

INPUT
LINE INPUT

INPUT allows the operator to type data into

one or more variables. The ASCII input command has the following format:

INPUT [#<file number>,] <I/O list>

where <file number> is the specification of where the input is coming from. Currently defined file numbers for input are 0 for the console, and 1 for the reader. The file number may be any arbitrary expression that evaluates to a valid file number. If the file number clause is omitted, then the console is assumed. The <I/O list> is a list of variables specifying where to put the data. ASCII input from either device must follow the normal rules for the specified input type. This means that input from the reader must be delimited by commas or carriage returns, and strings containing special characters must be surrounded by quotes.

Example: 35 INPUT A,B

would cause the printing of a question mark on the console as a prompt to the operator to input two numbers separated by a comma. If the operator doesn't type enough data then BASIC responds with 2 question marks.

```
Example:  10 INPUT A,B,C
          RUN
          ?5
          ?? 7,5
          READY
```

would input the value 5 to the variable "A" and when the operator typed carriage return, Basic wanted more data and so responded with 2 question marks.

The input statement may be written so that a descriptive prompt is printed to tell the user what to type.

```
Example:  10 INPUT "TYPE A,B,C";A,B,C
          RUN
          TYPE A,B,C? (ans)5,6,7
          READY
```

This causes the message placed between the quotes to be typed before the question mark. Note the semicolon must be placed after the

last quote.

The entry of just a carriage return as the response to an INPUT statement does not return it to command level. The empty line will correspond to a single value of zero if a numeric variable was being input, or the null string variable was being input. If more variables are left in the input list, additional input will be requested. To terminate program execution and return to command level, a control-C should be entered just as if the program was running.

The LINE INPUT statement provides increased flexibility in handling console input. The format of the statement is:

LINE INPUT [#<file number>,] <I/O list>

where <file number> is the specification of where the input is coming from. Currently defined file numbers for input are 0 for the console, and 1 for the reader. The file number may be any arbitrary expression that evaluates to a valid file number. If the file number clause is omitted, then the console is assumed. The <I/O list> is a list of variables specifying where to put the data. ASCII input from either device must follow the normal rules for the specified input type. This means that input from the reader must be delimited by commas or carriage returns, and strings containing special characters must be surrounded by quotes.

The statement functions similarly to the normal INPUT statement. The prompt string, if present, is output to the console. Each variable in the input list must be a string variable and is assigned the value of the entire input line as typed by the user, with no formatting. If more than one variable is present, then additional lines are requested. An empty line (just a carriage-return) has the value of the null string.

PRINT/LPRINT PRINT and LPRINT are the converse of INPUT in that they print out data on the console and line printer. For an explanation of these

commands see GROUPS 3 and 4.

INP

Basic has the ability to directly read an input port. The INP function takes as its argument the number of the port to be read, and the result may be assigned to a variable or printed directly.

```
Example:  10 A=INP(0)
          20 PRINT INP(0)
```

would in both cases input from port zero. In line 10 the value input from the port is placed in variable "A" and in line 20 it is directly printed.

MLOAD

The input statement MLOAD is a high speed input mode for those users who have only non-controlled I/O devices. The format of the statement is:

```
MLOAD #<file number>, <matrix 1> [,<matrix 2>...]
```

The statement inputs an entire numeric matrix at one time, in binary. Each matrix is preceded by a binary data prompt. The speed of input is such that an uncontrolled device can be used. Each matrix must be defined prior to its use in the statement. Multi-dimensional arrays are stored and loaded in a sequence with the last subscript varying most rapidly. There is no requirement that a matrix be read back into the same size matrix it was written from, all correspondence is up to the user. A binary prompt is required before each array read however.

MSAVE

The output statement MSAVE is a high speed output mode for those users who have only non-controlled I/O devices. The format of the statement is:

```
MSAVE #<file number>,<matrix 1> [,<matrix 2>...]
```

This statement outputs an entire numeric (not string) matrix at one time, in binary. Each matrix is preceded by a data prompt. The speed of output is such that an uncontrolled

device may be used. Each matrix must be defined prior to its use in the statement. Multi-dimensional arrays are stored and loaded in a sequence with the last subscript varying most rapidly. There is no requirement that a matrix be read back into the same size matrix it was written from, all correspondence is up to the user. A binary prompt is required before each array read however.

OUT

This command causes Basic to output data directly to any output port. The OUT command has two parameters separated by a comma. The first parameter is the port number and the second parameter is the data to be output.

```
Example:  10 A=1
          20 B=7
          30 OUT A,B
          RUN
```

would cause a seven to be output to port one, and if your console data port is port one the bell would ring since a 7 is a BELL in ASCII code.

QUOTE

This statement is used for ASCII output devices. The format of this statement is:

QUOTE [#<file number>,] [<quote character>]

where <quote character> is either zero, omitted, or the decimal value of an ASCII character (as in the NULL command). If the value is zero (or omitted), then the output from the device will appear the same as in previous versions of Basic. If the value is non-zero, then the device will output in special QUOTE mode. In this mode, which only affects normal PRINT statements (not USING), all commas occurring in the I/O list of the PRINT statement will not cause the standard TAB function, but will be sent to the output device. In addition, any string variable printed will be preceeded and followed by the specified ASCII character (which is usually a double quote [decimal 34]). The effect of this mode is to provide output which is

directly readable by an INPUT statement.

Each of the three devices has defaults for each of the three characteristics. The console and list (0 and 2) devices have default widths of 72, null counts of 0, null characters of 0, and quote modes of 0. The punch device (1) has a default width of 253, null count of 0, null character of 0, and quote mode 34 (double quote).

WAIT

If you write a program to INP or OUT directly to the console for a purpose such as reading a paper tape from the teleprinter tape reader, Basic itself will interfere with inputting the data because Basic is looking at the console keyboard to see if a control-C is typed to abort execution or control-X is typed to return to the Monitor. The WAIT statement will place Basic in a loop, looking at a specified status port, until a specified condition occurs. Then and only then will the next statement be executed.

(NOTE: Be careful using this because it is possible to put Basic in a loop waiting for a condition that will never occur. Should this happen, your only recourse is to reset the machine, or examine the memory location 3 higher than the address the program was loaded at, and hit RUN again. Basic will then recover without destroying your program.)

```
Example:  100 WAIT A,B,C
          110 D=INP(A+1)
```

Basic will then input port "A", EXCLUSIVE OR the value with "C", and then AND the result with B. If a zero result occurs, then the process is repeated until a non-zero result occurs. Basic, in this example, will input from the next higher port and place the data in "D". The fact that at Line 110 Basic looked at the console port to see if the data was a control-C would not affect the proper inputting of data. In this example the status port is 0, the data port is 1, the data available bit is bit 2, and it goes low (0) to indicate that data is available on port 1. Then let A=0 for port Zero and One. Let B=4

to isolate Bit 2, and let C=255 so that a complement of the status occurs to follow the rule that data available is indicated by a non-zero result. If parameter C is omitted, then Basic defaults to zero for the value.

WRITE

As in the READ statement, this statement performs two functions. If no I/O list is present, a binary data header is written on the punch device. If an I/O list is present, the data is written as specified on the output device. All strings include their length as part of the data written.

The binary output statement has the form:

WRITE #<file number> [,<I/O list>]

PEEK

This function allows the direct retrieval of data anywhere in memory.

Example: 50 B=PEEK(A)

causes the value of the byte at address "A" to be assigned to the variable "B". Address "A" may range from 0 to 65535 (decimal).

POKE

Has two parameters.

Example: 57 POKE A,B

in which the first parameter specifies an address in which to insert the data specified by the second parameter. The address may range from 0 to 65535 and the data may range from 0 to 255.

COPY

The COPY command provides a method of moving, or duplicating, one section of a Basic program into another part of the program. The format of the command is:

COPY<new line>[,<increment>]=<line range>

The command copies the set of lines specified by <line range>, which is in the same format as the LIST command to that part of the program specified by <new line>. The lines are renumbered as they are copied, starting with <new line> and incrementing by <increment> (which is 10 if omitted). Before copying any lines, the new line numbers are validated to guarantee that they will not overlap any existing lines, and that the new lines do not fall with the range of the lines being copied. Only the line numbers are changed, the lines themselves are not modified. The original lines are also not modified.

EXCHANGE

To speed sorting operations, the EXCHANGE command has been added. The format of the statement is:

EXCHANGE <variable 1>,<variable 2>

This statement exchanges the values of the two variables specified. Both variables must be predefined, and of the same type. A single matrix element may be used as either of the variables. For example:

```
EXCHANGE A$,B$
EXCHANGE A$(2,3),C$
EXCHANGE C,D(I,J)
```

This exchange of values is accomplished in the most rapid way possible (strings just switch pointers).

3.6 GROUP 6 TRANSFER OF CONTROL AND RELATIONAL TESTS

EOF

User initiated end of file processing. The EOF statement may be used by itself to cause a software controlled initiation of the end of file processing. The format of the statement is:

EOF

This allows an end of file to be determined based on some programmed condition as well as on a hardware detected one.

GOTO

This statement followed by a valid existing line number will cause Basic to transfer control directly to that statement.

Example: 55 GOTO 100

will, if line 100 exists, cause execution of the program to resume at line 100.

GOSUB

This statement acts in a manner similar to that of GOTO except that the location of the next statement is saved so that a RETURN can be performed to return control.

RETURN

This statement is used to "return" control back to the statement following the most previous GOSUB that control came from.

ON EOF GOTO

End of data file detection. If an end of file indication is obtained from the Zapple Monitor, or a ctl-Z (1AH) is read in ASCII mode, then the end of the data file has been reached. If no end of file action is specified, an error message will be given. To allow the user to detect and process the end of file condition, the ON EOF GOTO command is used. The format of the command is:

ON EOF GOTO [<line number>]

If the line number is omitted, then the EOF processing is disabled. After the execution of this statement, any end of file encountered on a data input statement (INPUT,READ, or MLOAD) will cause a GOTO to the line specified. Execution will then continue with the statement at that line. This end of file processing is not a subroutine type of call, in that there is no way to "return" to the input statement causing the condition. Also, the occurrence of an end of file branch does not disable the ON EOF, so a subsequent end of file will again cause the GOTO to be executed.

ON x GOTO
ON x GOSUB

The ON statement causes control to be transferred to the "x"th line number in the list.

Example: 10 ON A GOTO 100,125,150

If A was equal to 1, control would be transferred to Line 100. If A=2, GOTO Line 125, etc. If A is equal to zero, or larger than the number of line numbers in the list, control will be given to the statement after the ON x GOTO. The value of x may range from 0 to 255.

CALL

A greatly improved method of invoking assembly language subroutines is provided by the CALL statement. The format of the statement is:

CALL <address>[,<argument 1>[,...,<argument n>]]

The <address> is an expression representing the machine address of the routine to call. The statement also allows the optional specification of arguments to be passed to the subroutine. Each argument expression is evaluated and converted to a 16-bit integer. These arguments are then pushed onto the stack, so the number of arguments is limited only by memory. When the subroutine is entered, the following information is available to it:

```
SP ->
     <argument n>
     <argument n-1>
          ...
     <argument 1>

HL ->
     <return address>

BC ->   # of arguments on stack
```

The arguments may then be popped off the stack in reverse order. In addition, note that the HL registers point to the location of the return address in the stack. If no arguments are desired, or some error abort is required, a SPHL followed by a RET will properly clean up the stack and return to the calling program. The BC registers contain the count of the arguments passed to the routine.

FOR,TO,STEP,NEXT

These keywords are used to set-up and control loops.

```
Example:  10 FOR A=B TO C STEP D
          20 PRINT A
          30 NEXT A
```

If B=0, C=10, and D=2, the statement at line 20 will be executed 6 times. The values of A that will be printed will be 0,2,4,6,8,10. "A" represents the name of the index or loop counter. The value of "B" is the starting value for the index, the value of "D" is the value to be added to the index. If D is omitted then the value defaults to 1. The "NEXT" keyword causes the value of "D" to be added to the index and then the index is tested against the value of C, the limit. If the index is less than or equal to the limit, control will be transferred back to the statement after the "FOR" statement. The index may be omitted from the "NEXT" statement, and if omitted the "NEXT" statement affects the most recent "FOR". This may be of concern in the case of nested "FOR-NEXT" statements.

```
Example:  10 DIM A(3,3)
          20 FOR B=1 to 3
```

```
30 PRINT "PLEASE TYPE LINE";B
40 FOR C=1 to 3
50 INPUT A(B,C)
60 NEXT C
70 PRINT "THANK YOU.";
80 NEXT B
```

IF,THEN,ELSE

The "IF" keyword sets up a conditional test.

Example: 25 IF A=75 THEN 30 ELSE 40

Upon execution of line 25 if A is equal to 75 then control is transferred to line 30. Else if A is not equal to 75 transfer control to line 40. The "THEN" clause may be replaced by a GOTO.

Example: 25 IF A=75 GOTO 30 ELSE 40

The THEN and ELSE clauses may contain imperative statements.

Example: 30 IF A=75 THEN A=0 ELSE A=A+1

The ELSE clause may be omitted in which case control passes to the next statement.

Example: 40 IF A=75 THEN A=0

Relational operators used in IF statements:

```
=  EQUAL
<> NOT EQUAL
<  LESS THAN
>  GREATER THAN
<= LESS THAN OR EQUAL
>= GREATER THAN OR EQUAL
```

The logical operators may also be used:

```
NOT Logical Negation
AND Logical And
OR  Logical Or
```

Example: 20 IF(A=0) OR NOT(B=4) THEN C=5

VARADR Variable address references. This function allows the actual address that a particular

variable resides at in memory to be obtained. The function format is:

VARADR (<variable>)

where <variable> is either a single variable or a matrix element reference. This function returns an integer value which is the address in memory at which the value of the variable or matrix element specified resides. This address may be used directly in the POKE and CALL statements, and in the PEEK function. The formats of the variable values are as follows:

Numbers: 6 bytes per number, least significant byte first. The first 5 bytes are the mantissa, with the sign in the 5th byte. The mantissa is stored in a sign/magnitude overnormalized form (the high order bit is always assumed to be 1, and is used to hold the sign bit). A sign bit of 1 is a negative value. The 6th byte contains the exponent, in excess 128 notation (the value is always positive = actual exponent + 128). It is a binary exponent. For example, the hex string 00 00 00 00 00 80 is the number .5.

Strings: 6 bytes per dope vector. The first byte contains the length of the string. The next byte is always zero. The next two bytes contain the address of the string itself, least significant byte first. The last two bytes are unused.

3.7 GROUP 7 TRIGONOMETRIC FUNCTIONS

ATN — Function to return the ARCTANGENT of a value. The result is expressed in radians.

Example: 10 B=ATN(.45)

returns the angle, expressed in radians, whose tangent is equal to .45.

COS — Function to return the COSINE of an angle, expressed in radians.

Example: 20 C=COS(A)

SIN — Function to return the SINE of an angle, expressed in radians.

Example: 30 D=SIN(A)

TAN — Function to return the TANGENT of an angle, expressed in radians.

Example: 40 T=TAN(A+B)

3.8 GROUP 8 MISCELLANEOUS FUNCTIONS

ABS

Function to return the absolute value of a value.

Example: 10 A=ABS(B+C)

If the result of the expression is positive then ABS returns that value. If the result is less than zero, then ABS returns the positive equivalent.

DEF,FN

These commands allow the user to define his own functions. A function defined in this way must have a name that begins with the letters "FN" followed by a valid variable name. For example "FNA", or "FNZ9". The function name is then followed by one parameter enclosed in parentheses. This parameter is a dummy argument and is included in the expression to the right of the equals sign.

Example: 159 DEF FNQ(X)=X*A

In this case A is a variable within the program and X is an argument that may be replaced with a constant or another variable when the function is used.

```
Example:  15 DEF FNQ(x)=X*A
           .
           .
          121 A=3
          122 B=4
          123 C=FNQ(B)+5
```

Thus Basic would take the argument B and multiply it by the value of A, add 5 to the product and place the result in variable "C".

User defined functions may return both numeric and string values. Also, functions may have more than one (or less - zero) parameters, and the parameters may be either numeric or string. Secondly, functions may now consist of more than one statement (multi-line functions). The format of a multi-statement function is as follows:

```
DEF FN<function name>[(<dummy 1>[,<dummy n>])]
        ...
   <function body>
        ...
  FNEND[<function value>]
```

The first line of the definition is the same as the first part of a single line definition. The difference is the absence of the equal sign following the header information. The function definition header is followed by the actual statements comprising the function body. The function body may consist of any valid BASIC statements except another multi-line function definition (single line definitions are valid). The last statement of the function must be the FNEND statement. This statement both signals the end of the definition itself, and, upon execution, returns the value of the function to the calling expression. The FNEND statement takes a single optional argument which may be any valid BASIC expression of the same type as the function itself (string or numeric). The value of the expression is the returned value of the function. If no expression is given, the function returns a zero if numeric, or the null string if string.

To allow for the programatic termination of the function, and for the return of alternate values, the FNRETURN statement is provided. This statement has the form.

```
FNRETURN [<function value>]
```

The FNRETURN statement, when executed, functions identically to the FNEND statement. It terminates the function and returns the specified value. The difference is that a function may have multiple FNRETURN's, and they may occur wherever a valid BASIC statement may occur in the function.

The following are a few examples of multi-line functions:

CALCULATE A FACTORIAL

```
100 DEF FNFAC(I)
200 IF I=0 THEN FNRETURN 1 'FAC(0)=1
300 FNEND FNFAC(I-1)*I 'FAC(I)=FAC(I-1)*I
```

BUILD A REPETITIVE STRING

```
100 DEF FNREP$(I$,I)
200 J$=""
300 IF I<=0 THEN FNRETURN J$
400 FOR J=1 TO I
500 J$=J$+I$
600 NEXT
700 FNEND J$
```

It should be noted that a multi-line function may call any other function (including itself), may do GOSUB's to anywhere in the program, and may modify any program variable.

EXP

This function returns the base of the natural log system "e" or 2.71828 raised to a power.

Example: 20 B=EXP(A)

If A is equal to 1 the result is "e" or 2.71828.

FRE

This function, when used with a dummy variable, returns the amount of memory available for Basic Programs and variables, but which is currently unused. If FRE is used with a dummy string variable, it returns the amount of currently unused string space.

Examples: 30 A=FRE(X)
40 B=FRE(X$)

INT

Returns the integer portion of a number. This is essentially a "round-down" operation. For negative arguments the result would be the next more negative integer.

Example: 50 C=INT(D)

If "D" has a value of 5.25 then "C" will have a result of 5.0. If "D" has a value of -3.4, then "C" will be set to -4.0.

LOG

Returns the natural logarithm of the expression used as an argument.

Example: 60 E=LOG(F+G)

will return the log to the base "e" of the expression "F+G".

SGN

Will return the value +1 if the argument is greater than zero, zero if the argument is zero, and -1 if the argument is less than zero.

Example: 70 H=SGN(I)

SQR

Returns the square root of the argument& The argument may not be less than zero.

Example: 80 J=SQR(K)

RND

Returns a pseudo-random number in the range between 0 and 1. The RND function uses a dummy argument to perform the following functions. An argument less than zero is used to initialize the pseudo-random number sequence. An argument of zero will return the previous random number. An argument of more than zero will return the next pseudo-random number in the sequence.

Example: 90 R=RND(1)

RANDOMIZE

Although this is not a function, it is discussed here because of its relation to RND. The RANDOMIZE command may be used to generate a truly random starting point for the pseudo-random number sequence. (RND)

3.9 GROUP 9 STRING RELATED FUNCTIONS

ASC

Returns the decimal number that represents the first ASCII character of the string expression used as an argument.

Example: 10 A=ASC(A$)

The decimal value 65 represents an ASCII "A". If the character "A" was the left-most character of string "A$" then variable "A" would be set to 65.

CHR$

Returns the ASCII character represented by the decimal value of the argument.

Example: 20 PRINT CHR$(7)

would ring the bell on the teleprinter connected to the console. A "7" is an ASCII bell.

LEFT$

Uses two arguments, the first is the string expression and the second is the number of characters to return from the left end of that string. The second parameter may range from 1 to 255.

Example: 30 B$=LEFT$(A$,5)

would set B$ equal to the first 5 characters of string "A$".

The two string functions LEFT$ and RIGHT$ will allow a length argument of zero, resulting in the return of the null string as the function value.

LEN

Returns the length of a string expression in bytes.

Example: 40 X=LEN(S$)

would set "X" to the number of bytes contained in the string "S$".

MID$ May have 3 parameters:
1) The string expression.
2) The position to start extracting characters.
3) The number of characters to extract. This value defaults to 1 if omitted.

Example: 50 A$=MID$(B$,5,6)

would take 6 characters starting at the fifth character from the left of string "B$", and place that string in A$.

The MID$ function can appear on the left-hand side of an equal sign to specify the insertion of a sub-string into an already existing string variable. The format is:

MID$(<string variable>,starting char>[,<length>])=<string value>

The <string value> will overlay the characters in the value of the <string variable> starting at the specified character (the first character is number one) for the specified length. If the length is omitted, the entire remainder of the string is replaced. If the string value is smaller than the length specified, it is padded to the specified length with trailing blanks. If it is longer, the extra characters are ignored.

For example:

```
A$="123456789"
MID$(A$,3,5)="ABCDE" => A$="12ABCDE89"
MID$(A$,3,5)="AB"    => A$="12AB   89"
MID$(A$,3,5)="ABCDEF"=> A$="12ABCDE89"
```

This is significantly faster than any method using splitting and reconcatenation of the strings.

RIGHT$ See LEFT$ except works on the right-hand end of the string.

STR$ Returns a string whose characters represent the numeric value of the argument.

Example: 70 A$=STR$(2.2)

would return the characters "2.2" preceded by a space as the result.

VAL Opposite of STR$. Returns the numeric value represented by a character string.

Example: 80 A=VAL("4.5)

would return the numeric result 4.5.

HEXADECIMAL CONSTANT

Hexadecimal constants may be directly used in the BASIC program. Each constant must be preceeded by an ampersand (&). The constant must not exceed 65536 (&FFFF) in value. These constants may be used anywhere a regular numeric constant (not a line number) may be used (including as an argument to the VAL function).

INSTR This function searches one string for a specified substring. The format of the function call is:

...INSTR(<string value>,<string value>[,<start char>[,<length>]])...

In the basic function call, the first <string value> is searched to see if it contains the second <string value>. If so, the value of the function is the character position of the first character of the matched string. If no match is found, the value is zero. The two optional arguments correspond exactly to the application of the MID$ function to the first <string value> prior to the search. If a match is found however, the resulting value is still relative to the first character of the string. For example:

```
INSTR("123456789","456") => 4
INSTR("123456789","654") => 0
INSTR("1234512345","34") => 3
INSTR("1234512345","34",6) => 8
INSTR("1234512345","34",6,2) => 0
```

3.10 GROUP 10 MISCELLANEOUS COMMANDS

END Stops the execution of the program. The END command may be placed anywhere in the program.

Example: 65520 END

REM Denotes that this line is a remark and is not processed.

Example: 10 REM This is a remark.

' (REMARK) Basic allows every statement to be followed by a remark. The remark is indicated by the use of an apostrophe (') preceeding it, and is terminated by either a colon (:) or the end of the line. For example:

```
100 I=1' INIT I:J=3*3'3 WORDS/ENTRY:GOSUB 234
'SETUP INFO
```

A statement consisting of just a remark is valid.

STOP Similar to the END command except that the message BREAK @ LINE (x) is printed, where "x" is the line number of the STOP command.

USR The USR command allows Basic to exit to a user provided assembly language routine, evaluate a value, and return with the result.

In use, Basic must be told where to go for the assembly language routine. When the USR function is referenced, Basic will call the USR transfer vector. Normally, this vector points to an error routine within Basic. In order to link to an assembly language routine, you must patch the address (start of Basic +6H) with a jump to your assembly language routine.

In your assembly language routine, in order to get the passed value, call 0027'H (start of Basic +27H). Basic will return with the passed value in registers D & E.

To return the result back to Basic, place the low byte of information in register B, and the high byte in register A, and call 002A'H (start of Basic plus 2AH).

To give control back to Basic, execute a RET instruction.

Having done the above, the Basic program and your routine can be made to interact at will by use of the USR function.

```
Example:  10 X=USR(Y)
          20 PRINT X
```

will pass the value "Y" to your assembly language routine. The returned value would be assigned to "X", and then printed on the console.

3.11 GROUP 11 COMMANDS TO HANDLE ASCII TEXT

ASAVE

The ASAVE command allows the punching of the ASCII text of a BASIC program. The format of the command is:

ASAVE

The command takes the entire current BASIC program and punches it in ASCII on the current punch device.

ALOAD,ALOAD*,AMERGE,AMERGE*

Four commands are used to allow the loading of ASCII source programs. These are ALOAD, ALOAD*, AMERGE, and AMERGE*. These commands all read ASCII text from the current reader device. Each incoming line must start with a line number, and terminate with a carriage-return. A line-feed immediately following the carriage-return is ignored, as are rubouts and nulls. Completely blank line (carriage return only) are also ignored. The input operation is terminated by either a control-Z in the text or by an EOF indication from the reader device. The two ALOAD commands clear the program storage area before starting, while the two MERGE commands merge the incoming lines with the existing program on a line number basic.

The difference between the Axxx commands and the Axxx* commands is the way in which they handle the reader device. The Axxx* commands assume a controlled reader, and stop after each line is read to convert the ASCII into internal format. The Axxx commands assume a non-controlled reader of any speed, and do not stop reading until the EOF is detected. These commands save the entire incoming ASCII text in memory prior to conversion to internal format, and hence require more memory for a given source than the Axxx* commands.

The format of the command is:

Axxx[*]

3.12 GROUP 12 SPECIAL FUNCTIONS & CONTROL-CHARACTERS

COMMA , Move to next TAB position or delimiter

SEMI-COLON ; 2 spaces between numbers

COLON : Used for multiple statements per line

RUBOUT The RUBOUT (DELETE) function echos the deleted characters bracketed by backslashes (\).

CONTROL-S,CONTROL-Q,CONTROL-C

During the execution of a BASIC program, the CTL-S and CTL-Q keys may be used to temporarily stop and then restart the program. The CTL-S key will stop the program, but will not echo or in any way affect any printed output. The CTL-Q key may then be used to resume the execution. A CTL-C may also be entered if it is desired to abort the execution and return to command level. Only CTL-Q and CTL-C will be recognized after a CTL-S.

CONTROL-U Delete input line.

CONTROL-X Return to the Monitor.

CONTROL-O Suppress the console output.

CONTROL-R A control-R entered whenever BASIC is accepting input (either while entering a program or entering data into a running program) will cause the current input buffer, with all rubouts and control-U's processed, to be typed out, and the input position to be left at the end of the line so more input may be entered. For example:

```
100 IFT\T\I-2\2-\=23 THEN GOSU\US\TO
123\32\32^R
```

would respond:

```
100 IF I=23 THEN GOTO 132
```

and leave the input pointer after the 132.

CONTROL-T While a program is running, a control-T may be entered on the console. This will result in the line number of the line currently being executed, being printed on the console. The program execution is unaffected by the use of this command.

3.13 GROUP 13 OPERATORS (listed in the order of evaluation)

A) Any expression enclosed in parentheses is evaluated from the innermost parenthesis first to the outermost parenthesis last.

B) ^ Exponentiation

C) (-) Negation. I.E. A minus sign placed so as to NOT indicate subtraction. For example:

A=-B or C=-(2*D)

D) * Symbol for multiplication. Used in the form 2*2(cr) yields an answer by Basic of "4".

E) / Symbol for division. Used in same manner as multiplication symbol.

F) + Symbol for addition. Example:

A+B (add B to A)

G) - Symbol for substraction. Example:

A-B (substract B from A)

H) RELATIONAL OPERATORS:

=	EQUALS
<>	NOT EQUAL
<	LESS THAN
>	GREATER THAN
<=	LESS THAN OR EQUAL TO
>=	GREATER THAN OR EQUAL TO

I) NOT Logical negation, such as A=NOT B

J) AND Logical AND

K) OR Logical OR

3.14 GROUP 14 ERROR HANDLING

ERL — Function used in error routine which allows it to process the error. This function requires no arguments (and hence no parentheses). The ERL function returns the line number at which the error occurred. A line number of 65535 indicates that the error occurred in a direct mode statement.

ERR — Function also used in error routine which allows it to process the error. This function requires no arguments (and hence no parentheses). The ERR function returns the number of the error which last occurred. A list of all defined error numbers is provided in Chapter 1, Section 1.3 of this manual.

ERROR — Software error generation. The ERROR command is provided to allow the use of software generated errors in conjunction with the error trapping capability. The format of the command is:

ERROR <error number>

where <error number> is any expression evaluating to an integer between 0 and 255. When this command is executed, an error occurs, and the value is stored as the error number. This number can either be one of the assigned error numbers (see list), or an arbitrary user defined number. If this command is executed, and no user error procedure is enabled (ON ERROR), then a normal program abort occurs. If the error number is defined, then the normal error message will be given. If not, the "UNKNOWN ERROR" message will be given.

ON ERROR GOTO — User error handling. The ON ERROR GOTO command is provided to specify a user error handling procedure. The command format is:

ON ERROR GOTO [<line number>]

where <line number> is the line number of the error handling routine. If the line number is omitted, then all user error trapping is disabled. After this statement is executed, any error occurring during the programs execution will cause a trap to the specified statement. This error routine has available two new functions, ERR and ERL, which allow it to process the error. See the descriptions of these functions under GROUP 14.

RESUME

After processing an error as required, the error routine returns to the regular program execution throught the RESUME statement. The format of this statement is:

RESUME [<line number>]

This statement is similar to the RETURN statement after a GOSUB, and in fact is nested in the same way. Every error trapping routine must eventually execute a RESUME statement. The RESUME statement with no line number re-executes the statement originally causing the error. The RESUME statement with a line number resumes execution at the specified line.

Since all error traps are nested in the same way as GOSUBS and function calls, it is possible for an error routine to begin with another ON ERROR statement, with its own error routine. In this case, each error routine must end by the execution of a RESUME statement.

It should be noted then when an error trap occurs, the effect of the ON ERROR statement which enabled the trap is disabled, and another error occurring prior to the execution of another ON ERROR statement will abort the program.

RESUME NEXT

This statement resumes execution at the statement (not the line) following the one causing the error. The format is: RESUME NEXT

CHAPTER 4

4.0 TDL BASIC VERSION 3 CAPABILITIES UNDER CP/M

A. Non-functional differences from Zapple version. The CP/M version of TDL Basic Version 3 works slightly different from the Zapple version due to the idiosyncracies of CP/M. For instance, CP/M always echos characters read from the console, which limits the interaction capabilities of programs. Specific differences are as follows:

1. The EDIT function always echos its commands. This may look slightly sloppy, but cannot be helped.

2. The rubout (DEL) function while typing in does not enclose the deleted input in backslashes (\), it merely echos the deleted characters.

3. The program break character is ctl-E rather than ctl-C. CP/M traps the ctl-C and exits to the operating system.

4. Basic V3 is loaded and executed by the command BASIC<cr>. A carriage return response to the "Memory Size?" message will properly assign available memory.

5. The ctl-X function is not available since there is no resident monitor in CP/M.

B. I/O differences. The CP/M version substitutes disk files for the Zapple reader/punch devices. All operations which could be performed to the reader or punch can now be performed to or from a currently assigned disk file. Only one input and one output disk file may be used at a time however.

C. Disk file assignment. Disk files may be dynamically assigned and deassigned to the input or output (reader or punch) function. This is done through the "OPEN" and "CLOSE" commands. The assignment is done through the following command:

OPEN #<file number>,<direction>,<file name>

For example:

OPEN #1,"I","PGM"

would open for reading a CP/M disk file called PGM.BAS.

In this, as in all following commands, the <file number> must be an expression which evaluates to 1, the only valid disk file number in this version. The <direction> must be a string expression which evaluates to either I or O (input or output), and the <file name> must be a string expression which evaluates to a standard CP/M file identifier with optional disk name and file extension. The disk name defaults to the logged in disk, and the file extension defaults to "BAS". This command assigns the input or output function to the specified disk file. It is an error to assign the input function to a non-existant disk file, or the output function to an existing one.

To deassign a file, the statement is:

CLOSE #<file number>,<direction>

Example:

CLOSE #1,"O"

For the input function, this merely breaks the file/function association. For the output function, all remaining memory buffers are emptied, and the file is closed, permanently establishing it on disk. If Basic V3 is aborted before an output file is closed, the file may be lost.

D. Disk file utilities. A number of additional facilities are available to maintain disk files in the CP/M version of Basic V3.

To determine if a particular file is on disk, the function "LOOKUP" is used. Its format is:

LOOKUP(<file name>)

Example:

LOOKUP "PGM.BAS"

The function returns a true (-1) if the file exists, and a false (0) if it does not.

The statement:

ERASE <file name>

Example:

ERASE "PGM.BAS"

erases the specified file from disk.

The statement:

RENAME <file name 1>, <file name 2>

Example:

RENAME "OLD.BAS","NEW.BAS"

renames file1 to file2.

E. Returning to CP/M. The command:

EXIT

closes any open files, and exits back to CP/M.

4.1 CP/M ERROR MESSAGES

29 DIRECTORY FULL
30 EXTENSION ERROR
31 NO DISK SPACE
32 INPUT FILE NOT FOUND
33 NO INPUT FILE
34 NO OUTPUT FILE
35 DUPLICATE OUTPUT FILE
36 OUTPUT CLOSE ERROR
37 INVALID OPEN TYPE
38 INVALID FILE ID

In a constant effort to improve the quality and usefulness of its publications, Technical Design Labs, Inc. provides this page for user feedback. Your critical evaluations of this document is our only effective means of determining its serviceability. Please give specific page and line references where applicable.

ERRORS NOTED IN THIS PUBLICATION:

SUGGESTIONS FOR IMPROVING THIS PUBLICATION: (i.e. clarity, organization,convenience,accuracy,legibility.)

MISSING DOCUMENTATION: (i.e. completeness.)

Name----------Date----------

Street----------

City----------State----------Zip Code----------

All comments and suggestions become the property of TDL. Send to Technical Design Labs, Inc.
Dept. of Product Improvements
1101 State Road
Princeton, N. J. 08540

Please indicate in the space below if you wish a reply.

TDL Z-TEL: Z80 Text Editing Language

User's Manual

(Revision 1)

November 8, 1977

Written by

Evelyn J. Tate and Randall B. Enger

Table of Contents

Chapter I

Introduction to Z-TEL

Z-TEL is a TDL utility program designed to provide a powerful set of techniques for editing text files. In addition to providing standard text editing features such as adding text, deleting text, searching, and changing text, Z-TEL offers ways for moving blocks of text, for command iteration, for expression arguments to commands, for forward and backward searching, and for invoking command macros.

Not only does it contain these powerful editing features, Z-TEL is easy to use because it includes detailed error detection and a number of options for tailoring the program to the user's individual needs.

This manual has four chapters and several appendices. This is Chapter One. Chapter Two is provided as a guide to the first-time user, and explains the concepts upon which Z-TEL is based. This is NOT a complete description of the entire editor. Chapter Three elaborates on Chapter Two by examining a typical Z-TEL editing session in detail. Chapter Four is a reference guide for the experienced user; it contains a detailed description of all Z-TEL commands and their effects.

Error messages are given in Appendix A, and some more advanced examples are given in Appendix B. Appendix C contains a short summary of all Z-TEL commands; it is meant to be a "quick-glance" guide to Z-TEL. Additional appendices will be included for special versions which become available.

Chapter II

Overview of Z-TEL Operation

The model for much of Z-TEL is paper tape: the file being edited is viewed as a string of individual characters. Conceptually, a pointer exists pointing somewhere into this string or at one of the ends, and almost all of the editor commands act relative to this pointer. The pointer is always before or after a character. The characters to the RIGHT of the pointer are thought of as being in the "positive" direction, and the characters to the LEFT of the pointer are thought of as being in the "negative" direction.

A. Deleting characters

For example, to delete the character immediately to the right of the pointer one enters "1D" (or just "D", since in the absence of an argument, the value "1" is generally assumed); to delete the character immediately to the left of the pointer, one enters "-1D" (or just "-D").

The commands "D" and "-D" are instances of the "delete characters" command, which takes a numeric argument and deletes that many characters from the pointer position. Again, negative numbers refer to characters before the pointer and positive numbers refer to characters after the pointer.

B. Moving the pointer

The "C" command behaves similarly, but instead of deleting characters, this command moves the pointer position. The "C" command takes one numeric argument - positive or negative - and moves the pointer to the RIGHT if the argument is positive and to the LEFT if the argument is negative. For example, the command "-5C" will move the pointer position to the LEFT 5 places.

Another way to move the pointer is with the "J" or "jump" command. "J", like "C", takes one argument, but instead of interpreting this argument as a relative offset from the current pointer, the argument is treated as an absolute position in the file. Thus, "0J" will put the pointer at the beginning of the file, and "100J" will put the pointer after the 100th character in the file.

The pointer will not go past the end of the buffer in either direction, so it doesn't hurt to use an argument too big. This is true in general for all commands in Z-TEL: the

two ends of the file behave as infinite sources of characters (for character oriented commands) and infinite sources of lines (for line oriented commands).

C. Inserting characters

Characters can be inserted at the current pointer position. The characters are inserted between the character to the left of the pointer and the character to the right of the pointer. For example, the command "ihi there" when the pointer is between the "well " and "George" of the string "well George" will result in the new string "well hi there George". By convention, the pointer is left after the last character inserted, so that the two commands "ihi " and "ithere" (in the order given) are equivalent to the one command "ihi there".

D. Searching

A character string can be searched for using the "s" command. The "s" is followed by a string, which is compared to the characters in the file starting at the current pointer position. "S" also takes a numeric argument, the sign of which is the direction to search. If this number is positive (remember that the absence of an argument is the same as "+1") the editor compares the string to the characters to the right of the pointer looking for the specified occurrence of the string, and if this number is negative, characters to the left of the pointer are searched. Again by convention, the pointer is positioned after the rightmost character of the matched string -- if the search was successful. If the string couldn't be found, then the pointer isn't moved.

E. Lines

Although the file is treated as a string of characters, the notion of a "line" has been added for convenience. A "line" is merely the string of characters between two linefeed characters, not including the left linefeed but including the right one.

Some editor commands are line oriented. For instance, the "L" command moves the pointer a number of lines, positioning it always at the start of a line -- just after the previous line's linefeed character. The command "3L" moves the pointer to the right past three linefeed characters, or "down 3 lines". The ends of the file are treated as linefeeds when a line oriented command is looking for more linefeeds than exist. The command "3L" given when only one line exists in a file will position the pointer at the end of the file. No matter how hard you try, you can't move the pointer further than just past the ends of the file.

F. Files and paging

A "file" to the editor is an arbitrarily long string of ASCII characters that is presented to it through the reader device. The beginning of a file is the first character presented to the editor through the reader; the end of the file is whenever a <control-z> is encountered in the input stream or when the monitor returns the carry flag set after a call on the reader device. The carry flag is returned by the monitor when the reader runs out of tape or otherwise stops for a fixed time period. A flag is set by the editor upon encountering one of these conditions; it can be reset by the user and more text can be read in if necessary, for example, in the case of spurious <control-z>'s in a file. (See the description of value register "E" in Chapter Four, Section D.)

The editing process generally consists of reading a file, making changes to it, and writing out a new file. All changes are made in the editor's file buffer kept in main memory. The <control-z> character is not considered part of the file itself. When one is read, the editor sets the "end of file" flag and stops reading. When the writing out of a file is finished, a <control-z> is appended to the new file by the editor so that end of file may be found when the file is read in again.

The editor can handle files much larger than the amount of main memory available to it for the file buffer. When this is the case, part of the file must be brought in, changed, and then written out so the next section can be read in. This process is loosely referred to as "paging" through the file. A "page" has no fixed length and is not delimited by any special character. Rather, what can fit in the file buffer, or what is currently in the file buffer at any given time is called a page. Paging through the file is a one-way proposition: once a page has been written out, it can be accessed again only by re-editing the file. (See the "EA" command described in Chapter Four.)

Some commands perform an action with respect to a "page", the same way some commands perform an action relative to "lines" (e.g. L, K) or relative to individual characters (e.g. C, D). The "yank" ("EY") command, for example, deletes the current page and reads in a new page. The looseness of the term "page" can cause problems if this command is used carelessly. The detailed descriptions in Chapter Four will describe which commands use "pages".

G. Basic syntax

The basic command structure is the same for all Z-TEL commands. In the following, square brackets denote an optional field, and angle brackets denote a class of similar items or numbers. All commands (with minor exceptions) fall into the following form:

[<n>[,<m>]][:][@][f]<command>[<reg-spec>][<string>]

where <n> and <m> are expressions and <command>, <reg-spec>, and <string> denote the class of commands, registers (to be discussed immediately), and strings, respectively. No one command takes all of the optional arguments, especially since <reg-spec> and <string> are mutually exclusive. The following is an explanation of each part of the syntax:

<command> -- one of the many commands described in Chapter Four. Most commands are single letters; some consist of two letters, the first of which is always "E".

<n> or <m> -- an expression. Expressions consist of numbers, variables, parentheses, and the following five operators: plus (+), minus (-), times (*), divide (/), and unary minus (-). The expression is evaluated using the "standard" precedence rules: "*" and "/" take precedence over "+" and "-"; unary minus has higher precedence than "*" and "/". Operators of equal precedence are evaluated left to right. Parentheses can be used to alter the standard precedence, and nesting of parentheses is provided but is limited to about 10 due to stack requirements. Variables -- value registers -- can be used in expressions in the same way numbers are. See Chapter Three for examples of the use of expressions.

<string> -- a string that either starts immediately after the command and terminates with the first escape character encountered, or starts after the first character after the command and terminates with the second occurrence of the first character. The "@" option invokes the alternate string specification. For example, the "S" (search) command will find (if one's there) the first occurrence to the right of the pointer of the string "abc" in each of the following cases: (1) "sabc$", (2) "@s/abc/", and (3) b@seabce".

<reg-spec> -- a single letter or number denoting either a text register or a value register. Text registers hold text and are denoted by the numbers "0" through "9". Value registers hold numbers between -65,535 and +65,535 and are denoted by numbers or letters. Some of the value registers have predefined meanings, and some of these are "read only" variables. Chapter Four explains these in depth. See also sections I and J below.

The "f" option -- signals that this command has two string arguments following. For example, "fsabcdef" is the search command with the two-string option in effect. Most commands that take two strings search for the first and replace it with the second. The above command finds

"abc", deletes it, and inserts "def" in its place. There is no requirement that the two strings be of equal length.

The ":" option -- means something different for each command, but usually indicates that something is to be deleted. An example is the "P" command, which puts text into a text register: if the ":" option is present, the text is deleted from the file buffer after it is copied into the text register, whereas if the ":" is absent, the text exists in both places after the command. See the detailed descriptions in Chapter Four for the effect that the ":" option has on each command.

Any number of commands may be strung together to form a "command string" -- a list of commands that are to be executed in order until the end is reached or an error is encountered. A command string is not executed until two consecutive escapes are typed at the console, but the escapes are not considered part of the command string and are ignored during command execution.

H. Iteration

A command string can be executed repeatedly by enclosing it in angle brackets ("<" and ">"). The character "<" is a command that optionally takes a numeric argument (which must be positive) as the number of times the enclosed command string is to be executed. For example, the command string "20<I-$>" will insert the character "-" 20 times.

Angle brackets nest just as parentheses do, so command strings like the following are possible:

"20<I-$5<CI.$>0TT>"

which will (insert a dash followed by five (skipping one character and then inserting a dot) and then display the line without moving the pointer) all 20 times. Nesting depth is limited to five due to stack requirements.

With no argument to the "<", the editor assumes an argument of infinity (which really means 65,535). The command string "<IHI$>" will insert the string "HI" until the file buffer is full. Also, a minus argument or a zero argument is treated the same as no argument.

This is not as useless as it might seem at first glance, because certain situations terminate the iteration before the count goes to zero. A failed search, upon the execution of the closing ">", will cause the iteration to stop. For example, the command string "J<Sabc$0TT>" will print out every occurrence in the file buffer of the string "abc". When the ">" is executed after the search fails, the iteration will stop even though the command string was not executed 65,535 times. Notice that the "0TT" will be

executed whether the search succeeds or fails, producing a situation where the current line will always be printed once before the iteration terminates. The semicolon command is available to provide a more graceful stop.

The semicolon ";" command is closely related to angle brackets, and, in fact, it can't be used outside of an iteration. The semicolon command, when encountered in a command string, checks the result of the last search command in the iteration. If this search was successful, then the semicolon command does nothing. If the search failed, however, then the semicolon command terminates the iteration. Perhaps the most common use of this command is in a case like the following:

"<Sxxx$;CIyyy$0TT>"

which will find all occurrences of "xxx" and insert the string "yyy" one character after the last "x", and then print the line. If the semicolon were NOT included, the same action would take place UNTIL no more of the string "xxx" were found. Then, instead of stopping, an additional "yyy" would be inserted and the line printed. Then the iteration would stop, because - as mentioned above - iterations stop on failed searches. The semicolon command, then, acts as a kind of "guard" for the rest of the string.

I. Value registers

Value registers hold numeric values in the range -65,535 to +65,535. These registers can be set using the "W" command: for instance, "1W0" loads into value register 0 the value 1. Value registers can also be used as arguments to commands by using "V" to pull the value out. For instance, "55W5" followed by "V5*3<I*$>" will insert the character "*" 165 times.

Some value registers are used by the system for storing information about the file or the environment. The end-of-file status is always stored in value register "E" where 0 means no end-of-file and 1 means end-of-file (other values for "E" are undefined). Another example is the "X" value register, which contains the "eXtent" of the matched string after a successful search. Therefore, the command string "Sabc$-VXD" will delete the first occurrence of the string "abc". Section D of Chapter Four describes all the value registers.

J. Text registers

Text registers hold text. The amount of text they can hold is limited only by the size of available memory. The "P" command ("Put") is used to put text into a text register. It has two forms, one will put a number of lines into a text register (e.g., "5P3" will copy 5 lines starting at the current pointer position into text register 3), and

the other will put a number of characters into a text register (e.g., "0,9P2" will copy the first nine characters of the file into text register 2).

The "P" command, when the colon option is present, will delete from the file buffer those characters that have been "put" into the text register. For instance, "5P3" copies 5 lines into text register 3 and leaves them in the file buffer, whereas the command "5:P3" puts 5 lines into text register 3 and then deletes them from the file buffer.

The "G" command does the inverse of "P": "G" "gets" text from a text register. This command takes only the text register argument, which means only the whole text register can be "gotten". The text from the text register is copied into the file buffer at the current pointer position, and the pointer is moved to the right by the number of characters inserted, leaving it after the last character inserted.

The "G" command understands the colon, when present, to mean a request to delete the contents of the text register after the contents are copied into the file buffer.

K. Macros

The contents of a text register will be interpreted as a command string when the "M" command is used. "M" takes a text register and causes the text in that text register to be interpreted as editor commands. For example, if text register 1 contains the text "Sabc$20<I*$>0TTL" and the command "M1" is entered, then the above text string will be executed as if it had been entered from the console.

Macros are made even more useful by the "*" command which, when entered as the first character after the editor's prompt, copies the previous command into a text register. Using this command is an easy way of saving the retyping of a command. This sequence:

```
<command-string>
*<text-register>
M<text-register>
```

will cause the <command-string> to be executed again.

L. Input and output

Input and output are accomplished through the two commands "append lines" ("A") and "output lines" ("O"). Each takes a numeric argument, which is interpreted to be the number of lines to be input (for "A") or output (for "O").

For example, the command "50A" will append to the end of the file buffer fifty lines from the reader device, unless

either there's not room for fifty lines or there aren't fifty lines left in the file. The pointer position after the append command is unchanged.

For output, the "O" command will take the specified number of lines from the top of the file buffer and output them to the punch device. The pointer position is left at the beginning of the buffer after the output command.

Some commands will cause I/O to take place depending on the environment. Examples of these commands are "N" and "EN" (which are the "non-stop search" commands: these read in more text if the string is not in the current file buffer), the "EX" command (which terminates the editing session), the "Y" command (which "yanks" in a new page), among others.

M. Getting started

Z-TEL is a relocatable, "rom-able" program, which requires about 200-hex bytes of working storage starting at absolute 100-hex. The minimum loading address is 300. (i.e. When loading, type R,300) After the program is loaded, a "G300" to ZAPPLE will start the program. The program will ask for the low and high addresses that it's to use for its file buffer. If no addresses are given, then Z-TEL will use the larger of the space between its lower end and 200-hex or so (where the fixed working storage ends) and its upper bound and the highest address available (as ZAPPLE tells it). If addresses for the file buffer are given (and they must be in DECIMAL), then Z-TEL will use these as long as this buffer area doesn't overlap the program code. Also, it is the user's responsibility to ensure that the file buffer resides in write-able main memory. The editor will then display the size of the file buffer and issue a prompt character ("*"). At this time, the editor is ready to accept commands.

A note on the "escape" character should be mentioned here. For teletype-type devices which have no ESCape key, the ALTmode key should be used. Z-TEL will treat the ALTmode character like an ESCape, but only if the upper case flag is set. So, on teletype-type devices, we recommend that the first command to Z-TEL always be "1WU"; this needs to be terminated by 2 ESCapes, which are "control-shift-M" characters. After this, the ALTmode key will work like the ESCape key.

Chapter III

Some Basic Examples

This section shows how some of the basic Z-TEL commands might be used in an editing session. Our short source file contains poetry which needs some revision.

In the following examples, user-typed commands (in upper-case only) are preceded by the Z-TEL prompt character (an asterisk); the dollar-sign symbol represents an escape character.

We start the file edit process by loading Z-TEL and then loading the file with the "A" ("append") command. Z-TEL starts at address 300-hex, so the ZAPPLE command "G300" starts the editor. Z-TEL signs on and issues a prompt. The file can then be loaded using the "append" ("A") command.

When the first page of the file has been read in, Z-TEL will respond with another prompt. Here we might want to find out what's in the file buffer. The command "HT", which is equivalent to "0,ZT", prints out the entire buffer -- it types all characters from the first to the last.

```
*HT$$
To be or knot 2 B:
Whether 'tis nobler 'tis the mind to suffer
Or to take arms against a sea of troubles,
Alas! poor Yorick.  I knew him, Horatio;
A fellow of the bows and arrows of outraged fortune
etc.
etc.
etc.
```

The text pointer is initially at position zero -- that is, before the first character of the buffer. The first error which we will tackle is the "knot" in the first line (which is the current line).

```
*Sk$-D0TT$$
```

This command string does the following: searches for the character "k" (the pointer is left after the "k"), deletes the preceding character (which is the "k"), and types the entire line without moving the pointer (the "0T" types the line up to the pointer, and the "T" types the part of the line which is after the pointer). At the end of the command string the pointer is left right before the string "not".

```
To be or not 2 B:
*S2$-D2DIto be$0T$$
```

This command string finds the character "2" on the line, deletes it and the following two characters (which are " B"), inserts the string "to be", and leaves the pointer directly after the inserted string. Note that the "0T" types only that part of the text line which is before the pointer; since the line so far doesn't include a carriage return and line feed, the Z-TEL prompt is displayed on the same line as the typed text.

```
To be or not to be*CI  that is the question:$0LT$$
```

This command string (typed directly after the prompt character) moves the pointer forward one character to space over the colon in the text, and then inserts more text on the same line. The "0LT" moves the pointer to the beginning of the line and types out the entire line.

```
To be or not to be:  that is the question:
```

We now tackle the next line. The second occurrence of the word "'tis" is a mistake.

```
*LT$$
Whether 'tis nobler 'tis the mind to suffer
*2FS'tis$in$0LT$$
```

The "2FS" command finds the second occurrence of the first argument string ("'tis") and replaces it with the second string ("in").

```
Whether 'tis nobler in the mind to suffer
```

The next line is in fine shape, but the ones after that have some problems.

```
*4T$$
Whether 'tis nobler in the mind to suffer
Or to take arms against a sea of troubles,
Alas! poor Yorick.  I knew him, Horatio;
A fellow of the bows and arrows of outraged fortune
*2LSof the$VO,.-3KY$$
```

This command string moves the pointer down two lines, to point to the line beginning with "Alas". It searches for a string which marks the end of the text we wish to remove. The search leaves the pointer right after the word "the" (which we don't want to remove). Value register "O" contains the pointer position before the search (see chapter IV), and "." is the pointer position after the search. The command "VO,.-3K" deletes all characters from the old pointer position up to three characters before the new pointer position, thus sparing the word "the". The command "Y" is equivalent to "-TT", and types the preceding line and the now-current one.

```
Or to take arms against a sea of troubles,
the bows and arrows of outraged fortune
```

The "bows and arrows" line contains a few mistakes. Here we correct them and print out the results of what we've done so far.

```
*DIT$FSbows$slings$Soutraged$-DIous$0,.TT$$
```

This changes lower-case "t" to upper-case "T", "bows" to "slings" and "outraged" to "outrageous". The command "0,.T" is an example of the "T" command with two arguments; it means "type out everything between character position 0 (the beginning of the text buffer) and character position '.' (the current pointer position)". The second "T" command, with a default argument of +1, types out the remainder of the current line.

```
To be or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
Or to take arms against a sea of troubles,
The slings and arrows of outrageous fortune,
```

The fourth line should be the third, and vice versa. Rather than deleting a line and retyping it, we can move it elsewhere by using a text register.

```
*0L:P2-LG2HTd$$
```

This command string moves the pointer to the beginning of the current line (the one we want to move), "puts" one line (the default) into text register 2 (any other would do as well), moves the pointer back up a line, "gets" the text from the text register, and types out the entire buffer.

```
To be or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
etc.
etc.
etc.
```

We have yet to clean up the end of this.

```
*Setc.$0L.,ZKIAnd by opposing end them?
$-2T$$
```

This command string finds the first "etc.", deletes all text from the beginning of that line to the end of the text buffer (which is the three "etc."'s), inserts the correct phrase to complete the sentence (notice that we have included a carriage return and line feed in the inserted text string), and types out the last two lines of the file.

```
Or to take arms against a sea of troubles,
And by opposing end them?
```

One more change which we might want to make to our text is to double-space it rather than single-space it. This is easy to do with the iteration feature.

```
*0J<S
$;I
$>HT$$
```

This command string does the following. The pointer is positioned at the start of the text buffer. The text is searched for a carriage return; when one is found, a second carriage return is inserted after it. When the search fails because there are no more carriage returns in the text, the iteration is ended and the entire text buffer is typed.

To be or not to be: that is the question:

Whether 'tis nobler in the mind to suffer

The slings and arrows of outrageous fortune,

Or to take arms against a sea of troubles,

And by opposing end them?

To save our corrected file, we use the "EX" command, which writes out the contents of the text buffer (our now-edited file) back to the punch device, and returns control to the monitor.

```
*EX$$
```

Chapter IV

Detailed Description of Commands

A. Basic commands

A [<n>] A

The "append" command inputs <n> lines from the reader device and stores them at the end of the file buffer. The pointer position is unchanged. If there isn't enough room in the buffer for <n> more lines, as many characters as will fit are read in. A <control-z> in the input marks an end-of-file, and appending will stop. The <control-z> is not stored in the file buffer.

If <n> is zero, then the "A" command will read text until either the buffer is three-quarters full or an end-of-file situation is encountered. Negative arguments are not meaningful to the "A" command. If <n> is omitted, then a value of 1 is assumed.

B B <string>

Upon execution of the "branch" command, control is transferred to the first label (defined by "!" -- see the "!" command) in the current command string that matches the <string> argument to "B".

The "current command string" is defined to be the buffer or text register from which commands are currently being executed. This means that branching into or out of a macro can't happen, since only the labels in the "current command string" are searched.

C [<n>] C

This command moves the pointer <n> characters from the current position. If <n> is negative the pointer is moved backwards. If <n> is zero the pointer position is unchanged. If no argument is specified, n=1 is assumed ("-c" means "-1c").

D [<n>] D

This command deletes characters from the file buffer. If <n> is positive, the <n> characters immediately following the pointer position are deleted; if <n> is

negative, the <n> characters preceding the pointer position are deleted. If <n> is zero, nothing happens; if <n> is omitted, "1" is assumed ("-d" means "-1d").

E (not a command by itself)

This is used to create an "extended" command set; e.g., "ER", "EW". For specifics see the "extended commands" section.

F (not a command by itself)

This is used to modify the "S" (search) and "N" (non-stop search) commands. For specifics refer to those commands.

G [:] G <text-register>

This command gets the contents of the text register indicated by <text-register> (where <text-register> is a number from the set 0 - 9) and copies the contents into the file buffer immediately following the current pointer position. The pointer position is changed by the amount of text inserted, so the pointer points after the last "gotten" character. When modified by ":", the command causes the contents of the text register to be deleted after the get.

H (not a command)

This is not a command; it is the equivalent of "0,Z" and is used with commands which can take two arguments. E.g., "HT" is equivalent to "0,ZT" and will type out all characters between the first and the last character positions in the file buffer.

I [@] I<string>

This command inserts <string> into the file buffer at the current pointer position. Normally, the ESCape character signals the end of the string to be inserted. When "@" is used, an alternate string delimiter (other than escape) is used; this provides for the insertion of a single escape in the text. (Since a double escape always indicates the end of a command string, two "I" commands are required to insert two consecutive escapes.) The alternate string delimiter is the first character following the "i" and can be any character not used in <string> (including <space>). E.g., the command strings "ihello$", "@i/hello/", and "@iihelloi" all do the same thing.

J [<n>] J

The "jump" command puts the pointer after character position <n>, for positive <n>. If <n> is zero, the pointer will be put before the first character in the buffer. If <n> is negative, it is treated as a positive number with value (0 - <n>); e.g., "-1J" is equivalent to "1J". Note that J without an argument means "0J", not "1J".

K [<n> [,<m>]] K

If only one argument is supplied, <n> lines from the current pointer position will be "killed". When <n> is positive, "<n>K" will delete all characters from the current pointer position up to and including the <n>th following line feed character. When <n> is negative, "<n>K" will delete all characters from the character preceding the pointer position back to the beginning of the current line, plus the <n> preceding entire lines. "0K" will delete all characters on the current line which precede the current pointer position. When two arguments <n> and <m> are supplied, all characters following position <n> and up to and including position <m> will be deleted; the pointer will be left following position <n>. For example, "1,2K" will delete one character (which was in character position 2) and leave the pointer at position 1. If arguments are omitted, "1K" is assumed.

L [<n>] L

This command moves the pointer position by <n> lines. If <n> is positive, the pointer will be positioned after the <n>th following line feed. If <n> is zero, the pointer is moved to the beginning of the current line. If <n> is negative, the pointer is moved to the beginning of the <n>th previous line. If <n> is omitted, "1L" is assumed.

M M <text-register>

This is a macro invocation. The text in text register <text-register> is treated as a command string and executed. The contents of the text register are not changed. The following restrictions apply: a) any iterations or other commands must be wholly contained within the macro; likewise, iterations cannot start outside and finish inside a macro; b) no command can be executed in a macro which affects the contents of the text register in which the macro is stored.

N [<n>][@] [f] N<string> [;]

This is a non-stop search. It is similar to the "S" command (q.v.) But if the <string> is not found in the current buffer, the contents of the buffer are written out and another page is read in and searched. "N" will fail only when the last page has been read in and the <string> still hasn't been found. When "N" fails, the pointer is left following the last character of the page currently in the buffer (which is the last page of the file).

When the "f" modifier is present, the "N" command takes a second string which will replace the string found - after a successful search. Thus, the command string "<FNhi$hello$>" will replace all occurrences of the string "hi" with the string "hello" throughout the entire file (starting at the current pointer position, of course.)

The "N" command takes one numeric argument which, like the numeric argument to the "S" command, is the occurrence for which to search. Unlike the "S" command, however, the "N" command does not handle negative arguments.

O [<n>] O

This command outputs <n> lines from the beginning of the file buffer to the punch device and deletes them. If the file buffer contains fewer than <n> lines, the entire file buffer is written out and deleted. If <n> is omitted, 1 is assumed. The pointer is left at the top of the file buffer after the deletion of the output lines has occurred.

P [<n> [,<m>]] [:] P <text-register>

This command puts text from the file buffer into the text register specified by <text-register>. if ":" is used, the text is deleted from the buffer after it has been copied into the text register. Use of arguments is the same as for the "k" command (q.v.); "<N>p" will put the number of lines specified by <n>, while "<n>, <m> p" will put characters as specified by <n> and <m>. If the text register contained text at the time of the put, its previous contents are deleted and the new text copied in. The command "0,0P<text-register>" will delete the contents of register <text-register> and leave it empty.

Q (not defined)

R (not implemented)

S [<n>] [@] [f] S<string> [;]

This command searches the file buffer, to the right if <n> is positive and to the left if <n> is negative, for the <n>th occurrence of <string>. <N> must be a positive number; if it is omitted, 1 is assumed. If the <n>th occurrence of <string> is found, the pointer is left immediately following the last character in the <n>th occurrence of <string>. If the <n>th occurrence isn't found, the pointer is left after the last character in the last occurrence which was found (if the absolute value of n > 1) or at the same position as at the beginning of the search (if |n| = 1, or if |n| > 1 but no occurrences were found). "@" can be used to specify an alternate delimiter (other than the normal escape) to be used for <string>; use of "@" is described in the "I" command (q.v.). A semicolon can follow the search command only in an iteration; it means, "If found, continue with the command string, else jump outside the scope of the current iteration". E.g., the command string

"<2Shello$;Igeorge$> HT"

would insert the string "george" after every other occurrence of "hello" and print out the entire buffer when done. (Note: the semicolon need not immediately follow the "s" command, and, indeed, is a command in its own right. See "special character commands" section.)

When the "f" modifier is present, the "S" command takes a second string. After a successful search, the editor will replace the first string (the one found) with the second string. For example, "FSbad$worse$" will change the first occurrence in the file buffer of the string "bad" to the string "worse".

T [<n> [,<m>]] T

This command types <n> lines if one argument is specified, or all characters between positions <n> and <m> if two arguments are given. If <n> is negative, the previous <n> lines and all characters on the current line which precede the pointer position are typed. If <n> is zero, all characters on the current line which precede the pointer position are typed. If <n> is omitted, "1T" is assumed.

U (not defined)

V V <value-register>

This command pulls the value out of the specified value register. Valid value register names are 0 - 9 and a - z. Certain registers are read-only, containing system variables; these can be interrogated, but not set by the "W" command. A list of predefined value registers is given in section D of this chapter.

W <n> W <value-register>

This command puts the value <n> into the specified value register if the specified value register is not a read-only register. An error occurs if it is a read-only register. (See Section D below for a complete list of the value registers.)

X [@] X <string>

This command displays <string> on the console device when executed in a command string. This is useful especially in macros. "@" can be used to specify an alternate string delimiter (other than the escape character).

Y [<n>] Y

This command is shorthand for "-<n>T<n>T". If <n> is negative the sign is discarded.

Z (not a command)

This represents the number of characters in the file buffer. It is a value, not a command. E.g., "ZJ-T" positions the pointer after the last character in the buffer and prints out the preceding full line and any partial line.

B. Extended Command Set

EA EA

The "edit again" command terminates the editing of the current file, just like "EX", but does not exit back to the monitor. Instead, after the file is completely written out another prompt is given. At this point, all text registers and value registers are intact. If, when editing a large file, it is desired to move a large piece of code back to a page that has been written out already, then use "EA" instead of "EX". Follow this with the appropriate sequence of "append" commands to load the first part of the file and then with the "get" command to include the new text.

EC [:] EC <value-register>

The "extract character" command puts the number corresponding to the ASCII value of the character to the right of the current pointer position into the value register specified. If the colon option is invoked, the character is deleted from the buffer. If the pointer is at the bottom of the file buffer (i.e., "." = "Z"), then an error occurs.

EE EE <text register>

The "EE" command displays the entire contents of a text register onto the console device.

EF EF

The "end file" command closes the current output file by writing out a <control-z>, does any required name changing, and returns control to the editor for continued editing.

EI <n> EI

The "extended insert" command inserts the character whose ASCII representation is the number <n> into the file buffer at the current pointer position. The pointer is moved past the character inserted. <N> must be a number between 0 and 127 inclusive.

EK [:] EK <value-register>

The "keyboard" command provides a means for interactive command strings and macros. Upon execution of the "EK" command, Z-TEL will wait for one character from the console input device. When the character appears, its

ASCII representation is put into the specified value register.

The optional colon controls the echoing of the character. If the colon is present, the echoing is inhibited; without the colon, the character is echoed.

EL [<n>[,<m>]] EL

The "print to list device" command functions exactly like the "T" command except that the output goes to the list device instead of to the console device.

EN [<n>] [@] [f] EN <string>

The "extended non-stop search" command is much like the "N" command, but instead of writing out the current page of text, the command deletes the current page before reading the next page. This can be useful for splitting a file into several pieces or for extracting a piece of a file.

A specific occurrence of the <string> can be sought by including a numeric argument <n>. This number must be positive.

EQ EQ

The "quit editing" command traps back to the monitor without writing out any more of the file.

ES [<n>] ES <text-register>

The "extended search" command performs the same search operation as the "S" command, except the string to search for is in the specified text register instead of the input string. The same value registers are used here as are in the "S" command (i.e., the O, S, X value registers).

ET <n> ET

The "extended typeout" command displays the character whose ASCII representation is the decimal number <n>.

EV EV <text-register>

The "extended value of" command is much like "V" -- the value of the register is returned, except with the "EV" command the value is represented by the string (assumed to be decimal, optionally prefixed by a "-") found in the specified text register.

EX EX

The "exit" command is the normal method used to end an editing session. The input file (if any) is copied after the file buffer to the output file. When the file has been completely written out, control is passed back to the monitor.

EY EY

The "yank" command brings a new page into the file buffer after deleting the one already there. As mentioned in Chapter Two, "a page" is loosely used throughout this text, and is roughly equivalent to whatever is in the file buffer (i.e. main memory) or -- when doing I/O -- about half the total capacity of the file buffer.

C. Special-character Commands

= <n>[,<m>][:]=

The "=" command displays the number <n> on the console device. If a second argument <m> is present, then it is interpreted as a field length in which the number <n> is to be displayed right justified. For example, "VF,6=" would display something like " 2431". This feature is useful for outputting columns of numbers. If the number to be displayed (<n>) won't fit in the field specified, the number will be prefixed by the character "!" to indicate overflow. The colon is used to inhibit the output of the carriage return, linefeed sequence that normally follows the number. <m>, if present, must be between 0 and 256 exclusive.

\ <n>[,<m>][:]\

The "\" command is much like the "=" command, the main difference being that the number is inserted into the file buffer at the current pointer position instead of being displayed on the console device. As above, the second argument, <m>, denotes the size of the field in which the number will appear - again, right justified. If present, <m> must be between 0 and 256 exclusive. If the first number won't fit in the field as specified by the second number, then the first number will be prefixed by the character "!". The colon option for "\" is the inverse of the colon option for "=": if present with "\", the ":" causes a carriage return, linefeed sequence to be inserted after the number.

% [<n>]%<value-register>

The "%" command adds the number <n> to the contents of the specified value register and both stores the sum back into the value register and returns it to the next command. If <n> is missing, the value 1 is assumed (as with most commands). Negative values of <n> are permissible. For example, if value register one contains the number 55, then the command string "%1=" will add 1 (the default) to 55, storing the sum back into value register 1 and passing the sum (56) to the "=" command to be displayed on the console device.

? ?

The "?" command starts a command-by-command trace of the editor. (A further description of the trace feature will be forthcoming soon.)

# #

The "#" command resets the trace feature.

. .

The "." ("dot") command is equivalent to the number of characters to the left of the current pointer position -- that is, it is the numeric representation of the current pointer position. For example, after a successful search, the command "VS,.K" will delete the string that was just matched.

< [<n>]< .... >
> The "<" command signals the start of an iteration. The ">" command signals the close thereof. The command string enclosed by the "<", ">" pair is repeatedly executed -- <n> times if <n> is present, or "forever" if <n> is not. ("Forever" is defined to be 65,535.) The exception to this rule is when a search is included within the "<", ">" pair. For example, "35<%1\i. Hi there$>" will insert 35 numbered "Hi there"s into the file buffer. Searches, when failing, will cause the iteration to stop, when either a ";" or a ">" is reached. (See the discussion under the ";" command for more detail.) Iterations can be nested up to 5 levels. The pair "<", ">" must be wholly contained in a macro or outside a macro; splitting across a macro boundary is not recognized.

; ;

The ";" command changes the flow of control inside an iteration. This command checks the result of the most recent search command; if the search was successful, the ";" command does nothing. If the search wasn't successful, control is transferred to the matching ">" and the iteration is terminated. The ";" need not follow the search command immediately -- anytime later within the iteration will do.

* *<text-register>

When used as the first character of console input, the character "*" is interpreted as a command which puts the previous command string as text into the specified text register. The "*" character normally signifies the multiplication operator, and is interpreted as a command only when it is the first character in the input buffer when the input string is completed. If another character was typed, <rubout> or <control-u> or <control-e> can be used (see the description of these commands) to erase it; this special use of "*" can still be invoked. In fact, up to nine characters can

be typed and rubbed-out, as long as the "*" is the first character in the buffer when the string completes. Commands after the "*" in this command string are ignored.

, ,

The "," command is used to separate two arguments, and causes no overt action to take place. "<n>,<m>T", for example, is the means to pass two arguments to the "T" command.

<linefeed>

A linefeed character as the first character typed on the console device after a prompt is interpreted as the string "LT<escape><escape>". It is echoed as a "LT$$", and is executed immediately. It is included as a shorthand notation for a commonly used command.

<backspace>

A backspace character (control-h) as the first character typed on the console device after the prompt is interpreted as the string "-LT<escape><escape>". It is echoed as the string "-LT$$", and is executed immediately.

! [@]!<string>

The "!" command does nothing, but it is used by the "B" command to find where in the command string to branch to. The "B" command will search the command string from the beginning looking for a matching string prefixed by a "!". When found, the next command executed will be the command following the "!" command. Note that the "!" command provides a convenient way to document long macros, since the <string> is not used in any way by the editor. As usual, the "@" option provides an alternate way to specify the <string>.

Note that the "B" command will not look outside the current command string. In other words, if a "B" command is executed inside a macro, then only labels (the "!" commands) INSIDE the macro body are searched.

" <n>"<branch condition>

The """ command is a conditional branch command. The argument <n> is compared according to the specified <branch condition> and, if not safisfied, control is passed to the matching "'" character to the right. The different <branch condition> codes are as follows:

E - branch unless <n> = 0
N - branch unless <n> # 0
L - branch unless <n> < 0
G - branch unless <n> > 0
D - branch unless <n> is (in ASCII) a digit
A - branch unless <n> is (in ASCII) alphabetic
V - branch unless <n> is (in ASCII) lower case alphabetic
W - branch unless <n> is (in ASCII) upper case alphabetic

' '

The "'" command is the closing character for the "" command. When a condition is not satisfied for the "" command, then the editor searches for the matching "'". When found, control is passed to the command past the "'". The "'" command does nothing by itself; it merely marks a place in the command string. Notice that "" and "'" nest the same way "<" and ">" do, and matching requires being at the same level lexicographically.

<control-e>

The <control-e> character has two meanings: it can be used during the typing of a command string when it is desired to erase the entire string typed since the prompt, and it can be used when the editor is executing a command string, and it is desired to cause a break in the execution. This break occurs the next time a command is about to be executed. In either case, a prompt is given and the editor waits for the next input string from the console device. The current command string execution is terminated.

<control-f>

The <control-f> character causes the entire input string typed since the prompt to be retyped. The <control-f> is not inserted into the input buffer unless it is preceded by a <control-r> (see the description of this command). This is useful when many corrections have been made using <rubout> and <control-u> (again, see the description of these commands).

<control-o>

The <control-o> character stops output to the console without stopping the execution of a command string. A second <control-o> removes this inhibition. The inhibition is always removed when the editor is waiting for console input. This character is useful for leapfrogging through a long typeout: by hitting pairs

of <control-o>'s, one can skip over the displaying of groups of characters.

<control-r>

The <control-r> character causes the next character (with the exception of <control-x>) to be inserted into the command string input buffer without cny special effects of the next character occurring (e.g. <control-f>). A <control-r> itself may be inserted into the command string by prefixing it with another <control-r>.

<control-s>

(not yet inplemented)

<control-t>

The <control-t> character is much like the <control-f> character, but instead of causing the entire input string to be retyped, only the current line is retyped -- with any local corrections made, as above. The <control-t> is not inserted into the input buffer unless it is preceded by a <control-r>, in which case the <control-t> is treated as a normal character.

<control-u>

the <control-u> character is to the <control-e> character what the <control-t> character is to the <control-f> character. That is, <control-u> erases the current line of input. The <control-u> character is not inserted into the buffer unless it is preceded by a <control-r>, in which case the <control-u> is treated as a normal character.

<control-x>

The <control-x> character causes control to be passed back to the ZAPPLE monitor. As long as no registers and no memory locations are disturbed, editing can be resumed with a simple "G" command to ZAPPLE. This is useful when it is desired to assign a device for input or output (the ZAPPLE "A" command), for instance.

<rubout>

The <rubout> character causes the previous character in the input buffer to be echoed on the console device (unless the console device is a CRT -- covered next) and then deleted from the buffer. The <rubout>

character is not itself inserted into the input string. If the console device is a CRT, then the cursor is moved back one space, unless it is at the leftmost edge. The "T" value register controls whether the editor considers the console device to be a CRT or not. (See the section "Text and Value Registers", below.)

: (not a command by itself)

The ":" modifier is a general purpose command modifier, and is usually used to indicate that something should be deleted. For instance, "5P1" means "copy 5 lines into text register 1" and "5:P1" means "copy 5 lines to text register 1, and then delete them from the file buffer". See the individual command descriptions for the specific actions taken in each case.

@ (not a command by itself)

The "@" modifier can be used with commands that take strings to change the string delimiter from <escape> to anything else. For example, the command "@s/abc/0tt" will do the same thing as "sabc$0tt". The string delimiter in all cases is the first character after the command, and the string is closed by the second occurrence of that character.

D. Text and Value Registers

There are 36 value registers and 10 text registers. The value registers are denoted by "0" through "9" and "A" through "Z"; text registers are denoted by "0" through "9". Some of the value registers have predefined meanings -- values that are set by Z-TEL and sometimes by the user that affect the behavior of the editor or return interesting information. For example, the value register "L" is used by the system to control the number of characters displayed on a line. Thus "VL=" displays the current setting (it defaults to 72) and "80WL" sets it to 80.

Currently no text registers have predefined meanings.

The following describes all the predefined value registers.

A conversion inhibit flag

When value register "A" contains 0 (the default value), control characters encountered during output are converted by the editor into printable representations -- a "^" followed by a letter. When value register "A" has been set by the user to a non-zero value, this conversion is inhibited and characters are sent untranslated to the console device.

B cursor positioning

A special meaning has been attached to the "B" value register. This value register contains the current column that Z-TEL thinks the cursor or typing head is in. Since this value register is setable, one likely use for it is with graphics terminals, where certain characters are used to effect special features. Saving and restoring the "B" value register is a way to move the cursor and keep Z-TEL from a confused state.

C carriage-return enable flag

When value register "C" contains 0 (the default value), a double escape ends a command string. When value register "C" has been set by the user to a non-zero value, a carriage return is equivalent to double escape as a command string terminator. This may be useful with a terminal on which use of the escape is inconvenient. When carriage return enable is on, a carriage return character may be inserted into the text by preceding it with <control-r> in the command string. The linefeed character must be added explicitly in this

case. (Note: carriage return enable does not disable double escape as a command terminator. Double escape always acts as a command terminator.)

D duplex flag

When value register "D" contains 0 (the default value), the editor treats the console device as a full-duplex terminal. This can be changed by the user to a non-zero value, which indicates half- duplex. Note: this feature is not yet implemented.

E end-of-file flag

This is set to 1 by the editor when a <control-z> character has been encountered on reading characters from the reader device. This normally indicates end of the input file. When value register "E" is set to 1, the "A" (append) command does nothing. The user can reset this flag to 0 (for example, if he knows that the end of file condition was spurious, caused by garbage in the input file) so that further "A" commands can be done.

F free space

Value register "F" is a read-only value register which always contains the amount of free space left in the text buffer (in bytes); it is updated by the editor whenever characters are inserted or deleted. To inquire about the amount of empty file buffer space left, the command "VF=" can be used. This value, however, reflects a changing internal state, and it is consistently accurate for this purpose at the beginning of the execution of a command string only.

I iteration depth

Value register "I" is a read-only register which contains the current iteration depth. This will be zero outside an iteration and will be incremented by one for every "<" executed and decremented by one for every ">" executed.

L line length

Value register "L" is used to control the number of characters on a line; it defaults to 72. This can be changed by the user (for example, to accommodate a terminal with a shorter or longer line length) by using the "W" command.

N nulls required after a carriage return

Value register "N" is used to provide a delay for the physical carriage return when a carriage return character is sent to the console device. The default value is 3; increasing this will provide a longer delay for the carriage to return, and decreasing "N" will shorten this delay.

O old "dot" after a successful search

Value register "O" is a read-only value register which contains, after a successful search ("S", "N", etc.) command, the pointer position prior to the search. (Remember that the pointer is left after the matched string on a successful search.)

R lines read from the reader device

Value register "R" contains the number of lines read from the reader device from start of editing. This value register is a read-only register.

S string start position after a successful search

Value register "S" is a read-only register which contains, after a successful search, the pointer position of the start of the matched string. For example, "VS,.T" will print out the matched string if done immediately after a successful search.

T teletype/crt flag

Value register "T" contains the flag used by the editor to control the handling of <rubout>'s. The setting of this value register is done during initialization and depends on the monitor's setting for the console device: "T" is set to 0 for no assignment (which is assumed to be a teletype) or 1 for a CRT. In "teletype mode", the editor echoes the rubbed-out character. When set to 1, the editor will move the cursor back one position using the <backspace> character, erase the character just typed by printing a <space>, and then back up (since the <space> just moved the cursor again).

U upper case/upper-lower case flag

Value register "U" contains the flag used by the editor to control the handling of the output of lower case letters. If this register is 0 -- which it defaults to -- then no special provision is made for lower case letters. If this register contains a 1, then all lower

case letters (the ASCII range 61H to 7AH) are converted to upper case for console output. Lower case letters inserted into the file are not affected by this flag. Also, the non-zero setting provides a way to use the "altmode" key in the same fashion as the "escape" key. The escape character -- for those terminals with only an "altmode" key -- can be entered by typing "control-shift-K". This provides the means to set value register "U".

W lines written to the punch device

Value register "W" contains the number of lines written to the punch device from the start of editing. The "W" value register is a read-only register.

X extent of matched string after a successful search

Value register "X" contains, after a successful search, the length of the string that was just matched. The two command strings ".-VS=" and "VX=" are equivalent, so this register is merely a convenience. Value register "X" is a read-only register.

Y character to display for "Y" command

Value register "Y" -- when not zero -- is interpreted as the character to display between the two halves of the "Y" command. The "Y" command is defined to "-<n>T<n>T", but if value register "Y" is non-zero, then the "Y" command is defined to be "-<n>TVYET<n>T". (See the "V" and "ET" commands for more detail.) This can be used to insert a visual "marker" into the output of the "Y" command at the current pointer position.

Z total size of workspace

Value register "Z" is a read-only value register which contains the total size of the "in-core" file buffer. This is set up at editor initialization time and remains unchanged during the editing session. Users of the standard version will note that the limits provided for the file buffer will not equal the total size as shown in value register "Z". This is because part of that area is used for the editor's stack.

Appendix A

Error Messages

Most error messages display only a number in order to save room. This appendix explains all these numbers.

Error #	meaning
19	No room for insert. This can happen on an insert ("I") command as well as an "FS", a "\", and others.
30	Expression error. An error was detected in the evaluation of an arithmetic expression. This can be caused by mismatched parentheses, incomplete expressions, as well as a number of other things.
31	Expression error. Same as 30.
32	Missing value specifier. The editor either ran out of input or found a character that it couldn't use for the value register specifier. Value registers are denoted by the digits "0" through "9" and the letters "A" through "Z".
33	Literal too big. Numbers are restricted to the range -65,535 to +65,535.
34	Too many arguments. No command takes three arguments.
35	Too many arguments for this command. This command doesn't understand the number of arguments extant. For example, "HL" (which is equivalent to "0,ZL") is not understood.
36	Missing argument. This command insists on an argument and won't take a default like most other commands.
38	Unexpected end of input. The editor ran out of input during the parsing of a command. An argument with no command will cause this error.
39	Missing operand. An operator was left "dangling" during the evaluation of an

expression.

40 Missing operator. An operand was left "dangling" during the evaluation of an expression. For example, the command string "55V5=" will cause this error message.

41 Missing string delimiter. A character to match the first character after a string command used with the "@" option was not found.

42 Unknown "E" command. This command is undefined.

43 Unrecognized use of "F". Only a subset of the commands that take string arguments take an extra one. This command wasn't one of them.

51 Unknown command. This command is not in the editor's set.

52 Missing left angle bracket. A right angle bracket (">") was used without the editor seeing a matching left angle bracket ("<") first.

53 Iteration nested too deeply. Due to stack requirements of iteration, the nesting of angle brackets is limited to five.

54 Semicolon used outside an iteration. The semicolon (";") command has a meaning inside iterations only.

55 Macro calls nested too deeply. Due to the stack requirements of macro processing, the nesting of macro calls is limited to about five. This restriction includes recursive calls.

56 Missing right angle bracket. An iteration was started with a left angle bracket ("<") but the editor ran out of input without finding a matching right angle bracket (">"). Note that the command string will be executed once; the editor does not have a look-ahead feature.

57 Missing label. The editor tried to execute a branch command ("B") but couldn't find the appropriate label in the current environment.

58 Unknown conditional. The letter after the "conditional branch command" (the " command) was not in the set of known conditions.

59 Missing "'". A conditional branch was attempted, but no matching single quote could be found.

61 Negative argument to "O". The "output lines" command insists on positive arguments.

71 Bad value register specifier. The character given for the value register was not "0" through "9" nor "A" through "Z".

72 Protection error. This command attempted to change a read-only value register. Read-only registers are usually used internally by the editor and show up in the value register list because of the information they convey. Changing them is prevented because some untold side effects could occur.

101 Second argument too small. It turns out that almost all commands that take two arguments require the second one to be larger than -- or at least equal to -- the first. This error indicates that the command encountered a situation where the first was larger than the second.

115 Text registers in use by macros are "sacred". If a macro is executing from a particular text register, then that text register can't be deleted. This means a ":G" command and "P" command are disallowed for this text register.

116 No room for "put". The free space was smaller than the size of the text requested to be put into a text register.

117 No room for "put" or "get". This indicates a limitation of the current version in the handling of the case of ":P" and ":G".

118 No room for "get". There's not room enough in the work area for the size of text to be gotten.

122 Field length too big. The size field in which the number is to be positioned (right justified) is greater than 255. This is an arbitrary cut off point, but it was felt that a number larger than this is probably a typing mistake. This applies to the "=" and "\" commands.

123 Field length zero. The size field in which the number is to be positioned is zero. While provisions are made for handling overflow (the number is prefixed by a "!"), a

field length of zero is probably a mistake.

131 Bad text register specifier. The character used to specify the text register was not in the range "0" to "9".

137 Negative argument to a non-stop search. A backward search was requested of "N" or "EN". These two commands only work in the forward direction, mainly because paging itself works only in the forward direction.

139 Previous command not available. The special form of the "*" command was used when the previous command was overwritten prior to the entering of the "*" command. This can happen in several situations: more than nine characters were typed before the "*" was entered, or the space was needed by an insertion of text, or the buffer is so full that there's just no room.

171 Non-numeric text. The text in the specified text register can't be converted into a number because it's not all numbers. This error message only occurs when the "EV" command is used.

173 Non-ASCII character. The numeric argument to this command does not represent any ASCII character. This message only occurs when the "EI" command is used.

174 End of buffer reached. The "EC" command attempted to look at the next character in the buffer but found that the end of buffer had been reached, that there were no more characters.

Some error messages are textual and usually self-explanatory. They are included here for completeness.

"BUFFER ERROR"

The location specified during initialization for the file buffer overlaps the editor code space or the fixed location variables that start at location 100-hex.

"CAN'T FIND 'XXX' "

The specified string was not found.

"FOUND ONLY NN OF 'XXX' "

This message occurs when the string searched for was found, but not in the quantity requested. The command "4Sabc" when only 2 copies of the string are to the right of

the pointer will respond with "FOUND ONLY 2 OF 'abc' ". Note that the pointer will be left at the right of the last occurrence of the string that was found.

Appendix B

Advanced Examples

This appendix contains several examples of the lesser-used commands in Z-TEL. These examples, which use some of the more powerful commands, demonstrate why Z-TEL is indeed a "language" instead of just a text editor.

The first command string counts the lines in the buffer and to the current pointer position. Then, these values are displayed on the console.

```
0W10W2.W0          @!/zero counters, and save "dot"/
0J        @!/start at beginning of file buffer/
<@S/
/;%1>     @!/count all CR'LF strings/
V0J       @!/jump back to old "dot"/
<@S/
/;%2>     @!/count lines from "dot" to end/
V0J
          @!/display the counts:/
@X/LINE COUNT = /V1:=              @!/"V1" has total/
          @!/then the current if there are any lines/
V1"N@X/, CURRENT LINE = /v1-v2+1='
```

This next command string reverses the lines in the file buffer. The algorithm is as follows. The last line in the file buffer is put into a text register. This register is then unloaded at the top of the file, and the pointer position is recorded. Note that the pointer is left AFTER the last character gotten from the text register. At this point, if there are no more lines after the one we just got, we stop. Otherwise, we go get the (new) last line in the file buffer and continue.

```
0J        @!/start at beginning/
@!/LOOP/ @!/declare a label/
Z-."E@B/OUT/'
          @!/when "dot" = "Z", we branch out/
          @!/otherwise .../
.W0       @!/put pointer into valreg 0/
ZJ-L:P9   @!/put last line into textreg 9/
V0J       @!/go back to end of previously moved line/
G9        @!/get this line/
@B/LOOP/
@!/OUT/
0J        @!/and leave the pointer at 0/
```

This next command string converts all upper case characters in the file buffer to lower case letters. This takes advantage of the ASCII collating sequence in which all

lower case letters are 20-hex (or 32 decimal) higher than the corresponding upper case letters.

```
@!/LOOP/ @!/declare a label/
Z-."E@B/OUT/'
         @!/we're done when we're out of chars/
:EC4     @!/get ASCII of next char and delete it/
V4"W     @!/if it's upper case then ..n/
V4+32W4' @!/we add 32 to make it lower/
V4EI     @!/and insert it, whether changed or not/
         @!/just do one line for the example/
         @!/that is, stop on a CR (=13 decimal)/
V4-13"N@B/LOOP/'
@!/OUT/
```

Many enhancements could be made to this basic command string to make it "smarter" in its translation. One might be the following.

The next command string also converts upper case letters to lower case letters, but it will leave the first upper case character after a period as an upper case character instead of translating everything. We do this as follows. When a period comes along (ASCII representation is 46 decimal), we record this fact by setting value register 0 to 1. And just before translating a character from upper to lower case, we check this value register, and we bypass the translation if the value is 1.

```
1W0      @!/don't convert first char/
@!/LOOP/
         @!/test for being done/
Z-."E@B/OUT/'
:EC4     @!/get a char (and delete it)/
         @!/test for "."/
V4-46"E1W0'
         @!/now, is it upper case?/
V4"W     @!/do this if so/
         @!/first testing the bypass flag in 0/
V0"EV4+32W4'
         @!/don't translate if bypass flag is on/
0W0'     @!/reset it in any case/
V4EI     @!/insert the char, changed or not/
V4-13"N@B/LOOP/'
         @!/and stop on a carriage return/
@!/OUT/
```

The next example demonstrates the interactive features of Z-TEL. The command string accepts a string of decimal digits and outputs the number in its hexadecimal representation.

Assuming text register 2 has the following command string (which will be invoked as a macro below):

```
V3-9"G   @!/non-decimal ?/
V3+55ET  @!/display corresponding hex/
@B/OUT/'
```

```
     V3+48ET  @!/otherwise just print the number/
     @!/OUT/
```

then the following command string will do this conversion.

```
     @!/START/
     0W1      @!/register 1 accumulates number in/
     @X/
     -> /     @!/display a prompt/
     @!/LOOP/
     EK4      @!/get a character from console/
              @!/rubout gives another chance/
     V4-127"E@X/<RUBOUT>/@B/START/'
              @!/evaluate on a CR/
     V4-13"E@B/EVAL/'
              @!/if a digit, add it in/
     V4"D10*V1W1V4-48+V1W1@B/LOOP/'
              @!/no good otherwise/
     @X/ ??/@B/START/
     @!/EVAL/ @!/start of hex display routine/
     @X/
     /
     V1/4096W3
              @!/print first hex digit if non-zero/
     V3"NM2'  @!/textreg 2 does printing/
              @!/now shrink V1/
     V1-((V1/4096)*4096)W1
     V3W2     @!/save as zero-suppression flag/
              @!/and print second hex digit/
     V1/256W3
     V2+V3"NM2'
     V3+V2W2
              @!/shrink it again/
     V1-((V1/256)*256)W1
     V1/16W3
              @!/and print third hex digit/
     V2+V3"NM2V3+V2W2'
              @!/and once more .../
     V1-((V1/16)*16)W3
     M2       @!/print the last one/
     @X/h
     /
```

Appendix C

"Quick-Glance" Reference Guide

A. Basic Commands

A	"Append lines"	[<n>] A
B	"Branch"	B <string>
C	"move pointer by Character"	[<n>] C
D	"Delete character"	[<n>] D
E	(not a command by itself; see section B)	
F	(not a command by itself)	
G	"Get text from register"	[:] G <text-register>
H	(not a command; "H" is equivalent to "0,Z")	
I	"Insert string"	[@] I<string>
J	"Jump"	[<n>] J
K	"Kill lines or text"	[<n> [,<m>]] K
L	"move pointer by Lines"	[<n>] L
M	"invoke Macro"	M <tr>
N	"Non-stop search"	[<n>][@] [f] N<string> [;]
O	"Output lines"	[<n>] O
P	"Put text into register"	[<n> [,<m>]] [:] P <tr>
Q	(not defined)	
R	(not implemented)	
S	"Search"	[<n>][@][f]S<string>[;]
T	"Type text"	[<n> [,<m>]] T
U	(not defined)	
V	"Value of"	V <value-reg>
W	"set value"	<n> W <value-reg>
X	"display string"	[@] X <string>
Y	"display context"	<n> Y
Z	(not a command)	

B. Extended Commands

EA	"Edit Again"	EA
EC	"value of Character"	[:] EC <value-reg>
EE	"display contents of text register"	EE<text register>
EF	"End output File"	EF
EI	"Extended InsertWxZpD:f%Yh2V	
ES	"Extended Search"	[<n>] ES <text register>
ET	"Extended Type-out"	<n> ET
EV	"Extended Value of"	EV <text register>
EX	"Exit to Monitor"	EX
EY	"Yank"	EY

C. Special-character Commands

```
=     "display value"                <n>[,<m>][:]=
\     "insert value"                 <n>[,<m>][:]\
%     "increment register"           [<n>]%<value-reg>
?     set trace                      ?
#     reset trace                    #
.     "dot"                          .
<     iteration                      [<n>]< .... >
;     exit iteration on failed search   ;
*     "save previous command"        *<text-register>
,     argument separator             ,
<linefeed> "LT" if first character
<backspace> "-LT" if first character
!     "define label"                 [@]!<string>
"     conditional branch             <n> "<branch cond>
      E - branch unless <n>  = 0
      N - branch unless <n>  # 0
      L - branch unless <n>  < 0
      G - branch unless <n>  >  0
      D - branch unless <n>  is (in ASCII) a digit
      A - branch unless <n>  is (in ASCII) alphabetic
      V - branch unless <n>  is (in ASCII) lower case
          alphabetic
      W - branch unless <n>  is (in ASCII) upper case
          alphabetic
'     end of conditional branch      '
<control-e> erase entire input; break execution
<control-f> retype full input string
<control-o> inhibit output
<control-r> take next character as is
<control-t> retype current input line
<control-u> erase current input line
<control-x> trap to ZAPPLE
<rubout> erase previous character
:     (not a command by itself)
@     (not a command by itself)
```

D. Value Registers with Predefined Meanings

A	conversion inhibit flag
B	contains the cursor position
C	carriage-return enable flag
D	duplex flag
E	end-of-file flag
F	free space
I	iteration depth
L	line length
N	nulls required after a carriage return
O	old "dot" after a successful search
R	lines read from the reader device
S	string start position after a successful search
T	teletype/crt flag
U	upper case/upper-lower case flag
W	lines written to the punch device
X	extent of matched string after a successful search
Y	marker character for "Y" command
Z	total size of workspace

Appendix D

Additions for a CP/M Version of Z-TEL

To take advantage of the features offered by the CP/M operating system, several changes were made to Z-TEL. These changes include some new commands, some new error messages, and some minor changes to the existing command set to improve its capabilities in a disk environment. This appendix documents all these.

Under CP/M, Z-TEL is started by typing either "Z-TEL" or "Z-TEL <filename>". The second form will cause an automatic "EB<filename>" followed by a "0A", before the initial prompt is given.

During initialization, Z-TEL looks on the logged-in disk for a file with the special name of "INIT.TEL". If a file by this name is present, Z-TEL will read the file, put it into text register zero, and execute it as a macro before prompting the user. This feature offers a way to set flags and values, or do whatever else is required for an editing session. The file is expected to be just a string of normal Z-TEL commands. The exact sequence during initialization is as follows:

1. If "INIT.TEL" is on disk, then read its contents into text register 0.
2. If a file was specified to CP/M (i.e., "Z-TEL <file>") then an "EB" and "0A" are done.
3. The contents of text register 0 are executed as a macro.
4. A prompt is displayed.

A. New or modified commands

EB EB <disk string>

The "edit with backup" command opens the specified file for editing and opens a temporary output file. On closing (e.g. "EA" or "EX"), the input file is renamed to ".BAK" and the temporary output file is renamed to the input file name. An "ER" or "EW" command after an "EB" is permissible, but will cause the name changing here to be bypassed during closing.

Note: <disk string> ::=

[<dev1>:] <file name> [.<File ext>] [<dev2>:]

where <dev1> is the input device (if absent, the logged in device is assumed); where <dev2> is the output device (if absent, the input device name is assumed). <Dev2> is used only for the "EB" command; <dev1> is used for the output device in the "EW" command.

ED ED <disk string>

The "file delete" command deletes the specified file. This is useful when the disk is full and a file must be removed for editing to continue. (Alternatively, one can end the current file with the "EF" command and establish a new file on a different disk with the "EW" command.) Caution is urged with the use of this command. (Note: the syntax and description of <disk string> can be found under the "EB" command.)

EF EF

The "EF" command is changed slightly for the CP/M version of Z-TEL. The file is closed and a <control-z> is written out, as in the standard version. Then, before returning to the editor, any necessary name changing is done: the temporary file is changed from ".$$$" to the requested name after the previous file (if any) is renamed. See the description under "EX" below.

EO EO

The "edit over" command does everything the "EA" command does, with the addition of doing an "EB" on the new file -- if an "EB" was used before. If the "EB" command wasn't used, "EO" is the same as "EA". (Note: "EO" is currently implemented to be the "EA" command.)

EQ EQ

The "EQ" command hasn't changed for the CP/M version. It should be noted, however, that a temporary file (with the extension ".$$$") is left on disk after the command is executed.

ER ER <disk string>

The "edit read" command opens the specified file for input. (See note under the "EB" command for the syntax and description of <disk string>.) The "ER" command does not cause any data to be transferred into the text

buffer, the "A" (append) command must be used. The "ER" command also resets value register "R" to zero.

EW EW <disk string>

The "edit write" command opens the specified file for output. (See note under the "EB" command for details of <disk string>). The first device specified is used, not the second device specifier. The "EW" command resets value register "W" to zero.

EX EX

The "exit" command is the normal method used to end an editing session. If "EB" (as opposed to "ER" or "EW") was used to open the disk file, the previous backup file (if any) is deleted, the input file is renamed to ".BAK", and the temporary output file is renamed to the name of the original input file. A situation can arise where a file is found on disk with the same name as the one about to be used for the new output file. This situation arises when the output disk is different from the input disk for the "EB" command or when the string passed to the "EW" command represents a file already on disk. (Note that the "EW" command will not do anything with backup files.) The old file on the output disk is renamed to ".PRV" rather than ".BAK" since it hasn't been "updated" and isn't a backup copy of the file just edited. When all the name changing (if any) is done, control is returned to the monitor.

B. New value register descriptions

Value register "R" contains the number of lines read from the reader device. This meaning isn't changed by CP/M, but the register is reset to zero whenever an "ER" or "EB" command is executed. This is still a read-only register.

Value register "W" contains the number of lines written to the punch device from either the last "EB" or "EW" command. The "W" value register is a read only register.

C. New error messages

161 No file for input. The "A" ("append") routine was called but no file had been opened. (Files are opened using the "ER" and "EB" commands.)

162 No file for output . An "O" command ("output lines") or an "EX" command was issued (among a few others) without an output file having been opened first. Output files are opened with the "EW" and "EB" commands.

163 File not found. The file requested (by an "ER" command) was not on the specified disk.

"DIRECTORY FULL"

The specified disk has no room for the directory entry for this file.

"FILE EXTENSION ERROR"

The file name extension was unacceptable to CP/M.

"DISK FULL"

The output disk has no more room for the file. When this message appears during an editing session, the file should be closed (using the "EF" command) and a new output file should be opened on a different drive (using "EW") or a file on the output disk should be deleted using the "ED" command.

"FILE NOT FOUND"

The requested file is not on the specified disk.

"CLOSE ERROR"

This error shouldn't occur. If it does, either the editor made a mistake somewhere or the disk or drive was not write-enabled.

"[NEW FILE]"

This message means that the requested file was not found and a new file has been opened for output. This message will appear when the "EB" command is used to open a new file (including the implicit "EB" when starting up the editor from CP/M by typing "Z-TEL <filename>".)

"FILE NAME ERROR"

The file name specified was unacceptable to CP/M.

"REMEMBER: YOU HAVE ANOTHER 'XXX' "

If an "EW" command is used with a name for which a file already exists, this message will appear. On closing, the old file will be renamed to "<file>.PRV" to reflect its status as a previous copy. Since it's not a backup copy, it's not renamed to "<file>.BAK". The new file will then be renamed to the specified name.

In a constant effort to improve the quality and usefulness of its publications, Technical Design Labs, Inc. provides this page for user feedback. Your critical evaluations of this document is our only effective means of determining its serviceability. Please give specific page and line references where applicable.

ERRORS NOTED IN THIS PUBLICATION:

SUGGESTIONS FOR IMPROVING THIS PUBLICATION: (i.e. clarity, organization,convenience,accuracy,legibility.)

MISSING DOCUMENTATION: (i.e. completeness.)

Name----------Date----------
Street----------
City----------State----------Zip Code----------

All comments and suggestions become the property of TDL. Send to Technical Design Labs, Inc.
Dept. of Product Improvements
1101 State Road
Princeton, N. J. 08540

Please indicate in the space below if you wish a reply.

TDL Z80 Text Output Processor

User's Manual

Revision 1.0

February 14, 1977

Written by Neil J. Colvin

Introduction

Files to be processed by the TDL Text Output Processor are prepared by using the TDL Text Editor. The Processor accepts the prepared file from the reader device and produces a formatted document on the list device.

Each line read from the file is inspected for a first character of "." (period), which identifies a format control word. Format control words are not printed, but are interpreted to specify changes in the current output format. Control words may be entered in either upper or lower case, and should be separated from their operands, if any, by one or more blanks.

Control words may appear at the beginning of any line in the file, with any changes in format taking place below the point at which they occur. No input data should be included on lines containing control words, since this data could in some cases be lost or interpreted as an operand of the control word.

Control Words:

The control words are listed below.

Control	Meaning
.BL	Blank Line
.BM	Bottom Margin
.BR	Break
.BK	Break Mode
.CE	Center
.CM	Comment
.CO	Concatenate
.CP	Conditional Page
.DS	Double Space
.EN	End
.FO	Format
.HE	Heading
.HM	Heading Margin
.IG	Ignore
.IB	Ignore Break
.IN	Indent
.JU	Justify
.LL	Line Length
.NB	No Break
.NC	No Concatenate
.NF	No Format
.NJ	No Justify
.NS	No Space
.OF	Offset
.PA	Page
.PL	Page Length
.PW	Page Width
.PI	Permanent Indentation
.SS	Single Space
.SC	Space Character
.SP	Space Line
.SH	Subheading
.TB	Tab Settings
.TM	Top Margin
.UN	Undent

Default Values for Control Words

When processing a file, many variables, such as line length and page length, are assumed to have certain values ("default" values) until specified by a control word. This fact can simplify the formatting of files; many control words need not be used. The following is a list of control words having default values which are assumed to be in every file.

```
.PW 60
.LL 60
.PL 66
.TM 5
.BM 3
.HM 1
.PI 15
.TB 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
.FO  (.CO and .JU)
.SS
.BK
.SC _
```

Errors

The processor prints all errors encountered in the specified file with self-descriptive error messages, and with an exact image of the line at fault. Processing continues with the next input line.

Notes on Certain Uses

1. Top-Of-Page Format

Setting up the top of a processed page is not complicated once you understand the rules.

"Top Margin" is the number of lines between the top of

the page and the first line of text. It includes the heading line, the subheading line, and the heading margin.

A heading line and a subheading line are always printed, even if they are blank.

"Heading Margin" is the number of lines between the subheading line (even if it is not visible) and the first line of text.

For example, given the two control words

```
.tm 7
.hm 3
```

the following top-of-page format would be printed:

```
1st line:  Top margin blank line
2nd line:  Top margin blank line
3rd line:  Heading blank line with   Page <number>
4th line:  Subheading blank line
5th line:  Heading margin blank line
6th line:     "       "     "     "
7th line:     "       "     "     "
8th line:  First line of text
```

Given the default top margin (5) and the default heading margin (1) and the heading:

```
.he User's Manual
```

the following format would be produced:

```
1st line:  Top margin blank line
2nd line:  Top margin blank line
3rd line:  User's Manual          Page 92
4th line:  Subheading blank line
5th line:  Heading margin blank line
6th line:  First line of text
```

2. Space Character

A space indicates where line breaks (the end of a line) and justification (the insertion of spaces to make an even right margin) can occur. Sometimes it is desirable that a space be printed, but that a line end or extra

space insertion does not occur at this point. This "non-space" space is provided by the use of the "space-character". Normally, this character is the "_" (but may be changed by the ".SC" command.) When encountered by the processor, it is treated as a non-blank character, but is printed as a space. For example:

".IN_0"

This phrase will appear as ".IN 0"; it will not be split across two lines, and no extra spaces will be inserted between the N and the 0.

3. Tabulation

Tabulation is also provided through the use of a special character <HT> (Control-I). When encountered in the input, the tab character causes a "typewriter type" tabulation to the next "tab stop". The default is a tab stop every five spaces, but can be set to any value by the ".TB" control word. If the output pointer is currently at a tab stop, it will be moved to the next one (as with a typewriter).

It should be noted that a tab character or a space as the first character of an input line normally causes a "break", but this can be suppressed by the .NB and .IB controls.

Text Output Processor Example

```
.tm 10
.fo
.ce
Text Output Processor Example
.sp 2
.ds
This example will demonstrate some of the capabilities of
the Text Output Processor.  The next two pages will
appear first as they would in a file with the control
words visible; then the same file will appear as it
would if processed.
This paragraph was double-spaced with the .DS control.
.ss
.sp
No Break was needed here, since the .SS (Single-space)
control acts as a break.  Although this is in Format
mode, tabular information can be included:
.sp
        SPACE          .SP    .sp
        SINGLE SPACE   .SS    .ss
        DOUBLE SPACE   .DS    .ds
.sp
The blanks beginning each line caused a Break each time,
and the lines were not concatenated.
.sp
Use of the Line Length control allows space to be left
within a page for figures or drawings.  It may take some
experimentation to find how many lines will fit alongside
a figure.
.ll 30
.sp
The new line length must take effect at a paragraph,
since it acts as a Break.  The switch back to standard
line length (60) is also a Break, and usually ends a
paragraph.
.ll 60
.nf
By switching out of Format Mode         CAPTION
and doing some justification
by eye, other effects can be obtained.  This also takes
some practice and experimentation.
.sp
.fo
PARAGRAPHS
.br
If no space follows a paragraph heading, and if the
paragraphs are not indented, a Break is necessary in
Format Mode to keep the heading line from being
justified.
     A few leading blanks are the easiest way to force a
break and separate paragraphs, as this one was done.
.sp
```

Example

The Center control is handy for small figures included in the text. A .CE in front of each line of the figure is necessary. Note that leading blanks count as characters when the line is centered.

```
.sp
.ce
---------------------------
.ce
!  FORMAT  !  EXAMPLE  n  !
.ce
!          !  E          !
.ce
---------------------------
.ce
Figure A
```

Text Output Processor Example

This example will demonstrate some of the capabilities of the Text Output Processor. The next two pages will appear first as they would in a file with the control words visible; then the same file will appear as it would if output by the Text Output Processor. This paragraph was double-spaced with the .DS control.

No Break was needed here, since the .SS (Single Space) control acts as a break. Although this is in Format mode, tabular information can be included:

SPACE	.SP	.sp
SINGLE SPACE	.SS	.ss
DOUBLE SPACE	.DS	.ds

The leading blanks caused a Break each time, and the lines were not concatenated.

Use of the Line Length control allows space to be left within a page for figures or drawings. It may take some experimentation to find how many lines will fit alongside a figure.

The new line length must
take effect at a paragraph,
since it acts as a Break.
The switch back to standard
line length (60) is also a
Break, and usually ends a
paragraph.
By switching out of Format Mode CAPTION
and doing some justification
by eye, other effects can be obtained. This also takes
some practice and experimentation.

PARAGRAPHS
If no space follows a paragraph heading, and if the paragraphs are not indented, a Break is necessary in Format Mode to keep the heading line from being justified.

A few leading blanks are the easiest way to force a break and separate paragraphs, as this one was done.

The Center control is handy for small figures included in the text. A .CE in front of each line of the figure is necessary. Note that leading blanks count as characters when the line is centered.

FORMAT	EXAMPLE n
	E

Figure A

BLANK LINE Control

Purpose:

The BLANK LINE control word generates a specified number of blank lines before the next printed line.

Format:

.BL <n>

<n> specifies the number of blank lines to be inserted in the output. If omitted, 1 is assumed.

Usage:

The BLANK LINE control word may be used anywhere in the file to generate blank lines. If the end of the page is reached during a BLANK LINE operation, the operation is terminated and a new page is started. If, after the specified number of blank lines are inserted in the output, there are less than two printable lines remaining on that current page, a new page is started.

Notes:

This control word acts as a BREAK.

The printing of blank lines is independent of the current spacing.

Examples:

.BL 3

Three blank lines are inserted in the output before the next printed line.

.BL

A single blank line is inserted in the output.

BOTTOM MARGIN Control

Purpose:

The BOTTOM MARGIN control word specifies the number of lines to be skipped at the bottom of output pages, overriding the standard value of three.

Format:

.BM <n>

<n> specifies the number of lines to be skipped at the bottom of output pages. If omitted, 1 is assumed.

Usage:

This control overrides the standard bottom margin size of three lines, and need not be included in the file if that value is satisfactory. It may be included anywhere in the file, and the most recent value set applies on any page.

Note:

The BOTTOM MARGIN control word also acts as a BREAK.

Example:

.BM 10

Ten lines will be left blank at the bottom of the current page, if possible, and on all subsequent pages.

BREAK Control

Purpose:

When CONCATENATE is in effect, BREAK causes the previous line to be typed without filling in words from the next line.

Format:

.BR

Usage:

BREAK is used to prevent concatenation of lines such as paragraph headings or the last line of a paragraph. It causes the preceding line to be typed as a short line; the next line will be printed on a new line.

Notes:

Many of the other control words have the effect of a BREAK. No BREAK is necessary when one of these is present.

A leading blank or tab character on a line has the effect of a BREAK.

Example:

```
Heading:
.BR
First line of the paragraph . . .
```

This part of a file will be printed by SCRIPT as:

Heading:

First line of the paragraph . . .

If the BREAK control word were not included, it would be typed:

Heading: First line of the paragraph . . .

BREAK MODE Control

Purpose:

The BREAK MODE control reestablishes the BREAK function of the tab and space characters when they are the first on a line.

Format:

.BK

Usage:

This command is provided to reestablish the normal BREAK function initiated by the occurrence of a tab or space character at the start of an input line. This would only be necessary if a .NB command had been given previously.

CENTER Control

Purpose:

The line following the CENTER control word will be centered over the specified column.

Format:

.CE <n>

<n> specifies the column over which the following line will be centered. If omitted, the line length divided by two is assumed.

Usage:

The line to be centered is entered on the line following the CENTER control word. It starts at the left margin, and leading blanks will be considered part of its length. The column over which the line is to be centered is independent of any currently in effect controls (e.g. indent, undent, etc.).

Notes:

The CENTER control acts as a BREAK.

If the line to be centered exceeds the current line length value, it is truncated.

Examples:

```
.CE
Other Methods
```

"Other Methods" will be centered over the column whose number is equal to the current line length divided by two.

```
.CE 27
Column Title
```

When this line of the file is typed, the title "Column Title" will be centered over column 27 of the output.

COMMENT Control

Purpose:

The COMMENT control word causes the remainder of the line to be ignored, allowing comments which are not printed when the file is processed.

Format:

.CM <comments>

Usage:

The .CM control word allows comments to be stored in a file for future reference. These comments can be seen when editing the file or when the file is listed. The comments may also be used to store unique identification that can be useful when attempting to locate a specific region of the file during editing.

Example:

.CM Remember to change the date.

The line above will be seen when examining an unformatted listing of the file and remind the user to update the date used in the text.

CONCATENATE Control

Purpose:

CONCATENATE cancels a previous NO CONCATENATE control word, causing output lines to be formed by concatenating input lines and truncating at the nearest word to the specified line length.

Format:

.CO

Usage:

The CONCATENATE control specifies that output lines are to be formed by shifting words to or from the next input line. The resulting line will be as close to the specified line length as possible without exceeding it or splitting a word. This resembles normal typing output. This is the normal mode of operation for the processor. CONCATENATE is only included to cancel a previous NO CONCATENATE control word.

Note:

This control word acts as a BREAK.

Example:

```
.CO
Output from this point on in the file will be formed
to approach the right margin without exceeding it.
```

CONDITIONAL PAGE Control

Purpose:

The CONDITIONAL PAGE control word causes a new page to be started if space for less than the specified number of lines remain on the current page.

Format:

.CP <n>

<n> specifies the number of lines that must remain on the current page for additional lines to be printed on it.

Usage:

The .CP control word will cause printing to begin on a new page if "n" lines do not remain on the current page. This request is especially meaningful (1) before an .SP control word to guarantee that sufficient space remains on the current page for the number of spaces requested along with any titles, and (2) preceding a section heading to eliminate the possibility of a heading occurring as the last line of a page.

Note:

If no operand is specified with the .CP request, the request will be ignored.

Example:

.CP 10

If less than 10 lines remain on the current page, printing will begin on a new page. If 10 or more lines remain, printout will continue on the current page.

DOUBLE SPACE Control

Purpose:

The DOUBLE SPACE control word causes a line to be skipped between each line of printed output.

Format:

.DS

Usage:

DOUBLE SPACE may be included anywhere in the file to force double spaced output.

Notes:

This control word has the effect of a BREAK.

It affects all control words but BLANK LINE (.BL).

Example:

.DS

Blank lines will be inserted between output lines below this point in the file.

END Control

Purpose

The END control word is used to mark the end of the source file to the processor.

Format:

.EN

Usage:

The END control word must be the last line in the source file. When encountered, the output is spaced to the top of the next page, and the processor returns to the monitor.

Note:

This control word acts as a BREAK.

FORMAT Control

Purpose:

The FORMAT control word cancels a previous NO FORMAT control word (or NO CONCATENATE and/or NO JUSTIFY control word), causing concatenation and right justification of output lines to resume.

Format:

.FO

Usage:

The FORMAT control word is a shorthand way to specify the two control words: CONCATENATE and JUSTIFY. This control specifies that lines are to be formed by shifting words to or from the next line (concatenate) and padded with extra blanks to produce an even right margin (justify). Since this is the normal mode of operation for the processor, FORMAT is only included to cancel a previous NO FORMAT control word.

Notes:

This control word acts as a BREAK.

If a line without any blanks exceeds the current line length, it is truncated.

Example:

```
.FO
```

Output from this point on in the file will be formed to produce an even right margin on the output page.

HEADING Control

Purpose:

The HEADING control word specifies a heading line to be printed at the top of subsequent output pages.

Format:

.HE <line>

<line> specifies the heading to be printed at the top of subsequent pages.

Usage:

All of the line following the first blank after the HEADING control word is printed at the top of pages starting after the control word is encountered. No heading is printed on the first page of an output file. The heading is printed at the left margin. Its length must be at least 10 less than the output page width, to allow for a page number at the right margin. Leading blanks may be used to center the heading. The heading is printed in the line specified by the heading margin and top margin control words. Additional .HE control words may be included at any point in the file to change the heading on subsequent pages.

Note:

If a new heading is to be placed on a page forced

with the PAGE control word the HEADING control must precede the PAGE control.

Examples:

.HE ON-LINE EDITING SYSTEM

The characters "ON-LINE EDITING SYSTEM" will be printed at the left in the second-last line of the top margin on all pages started after this point in the file:

ON-LINE EDITING SYSTEM PAGE 7

.HE EDL

The heading blanks are considered part of the heading, so the characters "EDL" will be centered in the heading line:

EDL PAGE 8

HEADING MARGIN Control

Purpose:

The HEADING MARGIN control word specifies the number of lines to be skipped between the two heading lines and the first line of text, overriding the standard value of one.

Format:

.HM <n>

<n> specifies the number of lines to be skipped after the heading lines.

Usage:

The heading lines will be placed a specified number of lines above the first line of text. If no HEADING MARGIN control word is included in the file, the default value is one. The HEADING MARGIN specified must always be less than or equal to the current TOP MARGIN minus the two heading lines.

Note:

This control word acts as a BREAK.

Examples:

.HM 3

Three lines will be left between the heading lines and the first line of text. If the default top

margin of 5 is in effect, the headings will occur at the top of the paper followed by three more blank lines (the heading margin) and then the text.

```
.HM 1
```

The standard heading margin of one is set.

IGNORE Control

Purpose:

The IGNORE control word allows a line beginning with a period (.) to be printed.

Format:

.IG

Usage:

The .IG control word specifies that the following line of the input is to be treated as text even if it begins with a period.

Note:

This control has no other effect than the above. It is not a BREAK.

Example:

```
.IG
. . . . and so forth.
```

The second line will be treated just as if it did not begin with a period.

IGNORE BREAK Control

Purpose:

IGNORE BREAK causes the next break initiated by a leading blank or tab character to be ignored.

Format:

.IB

Usage:

A line beginning with a space or a tab character causes a break; the line is printed on a new line. When the first character of a line is a space or a tab character, IGNORE BREAK causes the break to be ignored. The two-line string of characters (the preceeding and the current lines together) is printed as one line.

Note:

Ignore Break control will affect the next space- or tab character-initiated break; it need not be placed immediately before it.

INDENT Control

Purpose:

The INDENT control word allows the left side of the printout to be indented.

Format:

.IN <n>

<n> specifies the number of spaces to be indented. If omitted, indentation will revert to the absolute margin.

Usage:

The .IN control word causes printout to be indented "n" spaces from the absolute left margin. This indentation remains in effect for all following lines, including new paragraphs and pages, until another .IN control word is encountered. ".IN 0" will cancel the indentation, and printout will continue at the absolute left margin.

Notes:

A) The .IN request acts as a BREAK.

B) The .IN request will reset the effective left margin, causing any .OF setting to be cleared. The .OF request may be used alone or in conjunction with .IN. When the latter is the case, .IN settings will

take precedence.

Examples:

.IN 5

All lines printed after this request will be indented 5 spaces from the absolute left margin. This indentation will continue until another .IN control word is encountered.

.IN 0

The effect of any current indentation will be canceled and printout will continue at the absolute left margin.

JUSTIFY Control

Purpose:

The JUSTIFY control word cancels a previous NO JUSTIFY control word (or part of a NO FORMAT control word), causing right justification of output lines to resume.

Format:

.JU

Usage:

This control word specifies that lines are to be justified (printed evenly on the right margin) by padding with extra blanks. If concatenate mode is in effect, the concatenation process occurs before justification. Since this is the normal mode of operation for the processor, JUSTIFY is only included to cancel a previous NO JUSTIFY control word or the NO JUSTIFY part of a NO FORMAT control word.

Notes:

A) This control acts as a BREAK.

B) If a line exceeds the current line length and CONCATENATE mode is not in effect, the line is printed as is.

C) This control word is seldom used without CONCATENATE mode. FORMAT should be used to enter both JUSTIFY and CONCATENATE modes.

Example:

.JU

Output from this point on in the file will be padded to produce an even right margin on the output page as long as the input lines do not exceed the line length.

LINE LENGTH Control

Purpose:

The LINE LENGTH control word specifies a line length that is to override the standard line length of 60 characters.

Format:

.LL <n>

<n> specifies output line length not greater than 132 characters.

Usage:

The LINE LENGTH control sets the length for output lines until the next LINE LENGTH control word is encountered. If no LINE LENGTH control is included in a file, the standard line length of 60 characters is used. In the JUSTIFY/NO CONCATENATE mode, lines shorter than line length are justified to length by blank padding. In the CONCATENATE mode, lines longer than line length are spilled into the following line. Shorter lines get words from previous or following lines to approach line length.

Note:

This control acts as a BREAK.

Example:

.LL 50

Succeeding lines will be no more than 50 characters in length.

NO BREAK Control

Purpose:

NO BREAK causes all subsequent space- or tab character-initiated breaks to be ignored.

Format:

.NB

Usage:

After the No Break control word, all breaks caused by a space or tab character as the first character on a line are ignored. Such lines will be printed where the previous line stopped.

Note:

NO BREAK will remain in effect until counteracted by .BK, Break Mode.

NO CONCATENATE Control

Purpose:

The NO CONCATENATE control stops words from shifting to or from the next line.

Format:

.NC

Usage:

The NO CONCATENATE control word stops words from shifting to and from the next line to even out the line length. The printed lines will appear as they do in the source file. It is useful for sections of files containing tabular information or other special formats.

Note:

This control acts as a BREAK.

Example:

.NC
Concatenation will be completed for the preceding line or lines, but following lines will be printed without words being moved to and from lines.

NO FORMAT Control

Purpose:

The NO FORMAT control stops the CONCATENATE and JUSTIFY mode, causing lines to be printed just as they appear in the file.

Format:

.NF

Usage:

The NO FORMAT control is a short-hand way to specify the two control words: NO CONCATENATE and NO JUSTIFY. This stops line justification and concatenation until a FORMAT, JUSTIFY, or CONCATENATE control word is encountered. It is useful for sections of files containing tabular information or other special formats.

Note:

This control acts as a BREAK.

Example:

.NF

Justification and concatenation will be completed for the preceding line or lines, but following lines will be printed exactly as they appear in the file.

NO JUSTIFY Control

Purpose:

The NO JUSTIFY control stops padding lines to cause right justification of output lines.

Format:

.NJ

Usage:

The NO JUSTIFY control word stops the padding of lines with additional blanks to form even right margins. If CONCATENATE mode is in effect, lines will be formed that approach the current line length but will not be forced to the exact length. The resulting lines resemble the output usually produced by a typist.

Note:

This control acts as a BREAK.

Example:

.NJ

Justification will be completed for the preceding line or lines, but following lines will be printed without additional blanks inserted to pad the line.

NO SPACE Control

Purpose:

The NO SPACE control withholds the printing of a space after the last word on a line.

Format:

.NS

Usage:

A space is normally printed after the last character on a line as it appears in the source file. The NO SPACE control causes the next line of text to be printed without this last space. It is used in creating longer lines when this space is not wanted.

Note:

NO SPACE will affect the next line of text; it need not be placed immediately before it. It affects only that line.

OFFSET Control

Purpose:

The OFFSET control word provides a technique for indenting all but the first line of a section.

Format:

.OF <n>

<n> specifies the number of spaces to be indented after the next line is printed.

Usage:

The .OF control word may be used to indent the left side of the printout. Its effect does not take place until after next line is printed, and the indentation will remain in effect until a break or until another .OF word is encountered. The .OF control may be used within a section which is also indented with the .IN control. Note that .IN settings take precedence over .OF, however, and any .IN request will cause a previous offset to be cleared. If is desired to start a new section with the same offset as the previous section; it is necessary to repeat the ".OF n" request.

Notes:

A) This control acts as a BREAK.

B) Two OFFSET control words without an intervening text line is considered an error condition.

Examples:

.OF 10

The line immediately following the .OF control word will be printed at the current left margin. All lines thereafter (until the next break or .OF request) will be indented 10 spaces from the current margin setting.

.OF

The effect of any previous .OF request will be canceled, and all printout after the next line will continue at the current left margin setting.

PAGE Control

Purpose:

PAGE causes a new page to be started.

Format:

.PA <n>

<n> specifies the page number of the next page. If "n" is not specified, sequential page numbering is assumed.

Usage:

Whenever a PAGE control word is encountered, the rest of the current page is skipped. The paper is advanced to the next page, the heading and page number are typed, and output resumes with the line following the PAGE control word.

Notes:

A) This control acts as a BREAK.

B) If the heading, line length, or other format parameters are to be different on the new page, the appropriate control words must appear before the PAGE word.

Examples:

.PA

The rest of the current page will be skipped. The heading and page number will be printed in the top margin of the next page, and output will resume.

```
.PA 5
```

Regardless of the number of the current page, the rest of that page will be skipped, the heading and page number 5 will be printed in the top margin of the next page, and output will resume.

PAGE LENGTH Control

Purpose:

The PAGE LENGTH control word specifies the length of output pages in lines. The value specified overrides the standard page length of 66 lines.

Format:

.PL <n>

<n> specifies the length of output pages in lines.

Usage:

The PAGE LENGTH control word allows varying paper sizes to be used for output. It should not be used to print additional lines on a page. Use the .BM control word. If no PAGE LENGTH control word is included in a file, 66 is the default value. This is the correct size of standard typewriter paper (at six lines per inch). Page length may be changed anywhere in a file, with the change effective on the current page if possible.

Note:

This control word acts as a BREAK.

Example:

.PL 51

Page length is set to 51 lines.

PAGE WIDTH Control

Purpose:

The PAGE WIDTH control specifies the width of the output page in characters. The value specified overrides the standard page width of 60 characters.

Format:

.PW <n>

<n> specifies the width of the output page in characters.

Usage:

The PAGE WIDTH control word allows varying paper sizes to be used for output. It should not be used to control the length of the line printed on the page. Use the .LL control word. If no PAGE WIDTH control word is included in a file, 60 is the default value. This is the correct size of standard typewriter paper (at ten characters per inch) leaving a one and one-half inch margin at the left, and a one inch margin at the right. Page width may be changed anywhere in the file, with the change effective on the current page if possible.

Note:

This control word acts as a BREAK.

Example:

```
.PW 50
```

The page width is set to 50 characters.

PERMANENT INDENTATION Control

Purpose:

The PERMANENT INDENTATION control specifies the location of the left margin on the output page. The value specified overrides the standard margin of 15 characters (one and one-half inch).

Format:

.PI <n>

<n> specifies the size of the left margin in characters

Usage:

The PERMANENT INDENTATION control allows a fixed amount of space to be left at the left margin of the output page. If no PERMANENT INDENTATION control word is included in the file, a default value of 15 characters (one and one-half inch) is used. This is the normal margin for typewriter paper when the printer start at the extreme left of the paper.

Note:

This control word acts as a BREAK.

Example:

.PI 10

Permanent Indentation is set to 10 characters (one inch).

SINGLE SPACE Control

Purpose:

The SINGLE SPACE control word cancels a previous DOUBLE SPACE control word, and causes output to be single-spaced.

Format:

.SS

Usage:

Output following the SINGLE SPACE control word is single-spaced. Since this is the normal output format, SINGLE SPACE is only included in a file to cancel a previous DOUBLE SPACE control word.

Note:

This control word acts as a BREAK.

Example:

```
.SS
Single-spacing will resume below this point in the
file.
```

SPACE CHARACTER Control

Purpose:

Space Character control enables the user to change the space character from "_" to any other character.

Format:

.SC <character>

<character> specifies the next "space character".

Usage:

The Space Character control is used to change the space character "_" to any other character. This would be done when "_" should appear in the text. The space character is treated like a character in joining two words which the user wants printed on the same line, but it is printed as a space.

Example:

.SC !

This allows "_" to be printed out. "!" will be printed as a space.

SPACE LINE Control

Purpose:

The SPACE LINE control word generates a specified number of blank print lines before the next printed line.

Format:

.SP <n>

<n> specifies the number of blank print lines to be inserted in the output. If omitted, 1 is assumed.

Usage:

The SPACE LINE control word may be used anywhere in the file to generate blank print lines. If the end of the page is reached during a SPACE LINE operation, the operation is terminated and a new page is started. If after the specified number of blank printed lines are inserted in the output there are less than two printable lines remaining on that current page, a new page is started.

Notes:

A) This control acts as a BREAK.

B) The printing of blank lines is controlled by the current spacing, either single or double. In doublespace mode, ".SP" generated blank lines

alternate with ".DS" generated blank lines.

Examples:

.SP 3

Three blank print lines are inserted in the output before the next printed line.

.SP

A single blank print line is inserted in the output.

SUBHEADING Control

Purpose:

The SUBHEADING control word specifies a subheading line to be printed under the heading line on the top of subsequent output pages.

Format:

.SH <line>

<line> specifies the subheading to be printed under the heading line at the top of subsequent pages.

Usage:

All of the line following the first blank after the SUBHEADING control word is printed under the heading line of pages starting after the control word is encountered. No subheading is printed on the first page of an output file. The subheading is printed at the left margin. Leading blanks may be used to center the heading. The subheading is printed on the line specified by the heading margin and top margin control words. If no subheading is specified, a line of blanks is printed in its place. Additional .SH control words may be included at any point in the file to change the subheading on subsequent pages.

Note:

If a new subheading is to be placed on a page forced

with the PAGE control word, the SUBHEADING control must proceed the PAGE control.

Example:

```
.SH LINE EDITING COMMANDS
```

The characters "LINE EDITING COMMANDS" will be printed at the left margin underneath the heading line on the top of all pages started after this point in the file:

```
ON-LINE EDITING SYSTEM                          PAGE 7
LINE EDITING COMMANDS
```

TAB SETTING Control

Purpose:

The TAB SETTING control word specifies the tab stops to be assumed for the following lines when converting the TAB character into the appropriate number of spaces.

Format:

.TB <n(1) n(2) n(3) n(4) n(5)>

<n(i)> specifies the column location of the (i)th tab stop; the sequence must consist of values separated by one or more spaces.

Usage:

TAB characters entered into the file during EDIT file creation are expanded by the processor into one or more blanks to simulate the effect of a logical tab stop. The TAB SETTING control word specifies the locations of the logical tab stops. This overrides the default tab stops of 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75. A TAB SETTING control word without any tab stops specified, results in reversion to the default tab settings. This control word is useful for indenting the beginning of a paragraph (remember a TAB causes a paragraph BREAK) or for tabular information and diagrams.

Note:

This control word acts as a BREAK.

Examples:

.TB 10 20 30 40

Tab stops are interpreted as columns 10, 20, 30, and 40.

.TB

Tab stops will revert to default values of 5, 10, 15, etc.

TOP MARGIN Control

Purpose:

The TOP MARGIN control word specifies the number of lines between the text and the top of the page. This includes the heading line, the subheading line, and the heading margin.

Format:

.TM <n>

<n> specifies the number of lines to be skipped at the top of output pages. n must be two or greater.

Usage:

The specified number of lines will be left at the top of succeeding output pages before the first line of text. The page number and heading, if any, are placed within the top margin by the .HM control. If no TOP MARGIN control word is included in the file, the default value is five. The top margin specified must always be equal to or greater than the current heading margin plus two lines for the heading and subheading.

Notes:

To determine a top margin,

a) select a heading margin

b) add two lines (for the heading and subheading)

c) add the number of lines to be left blank above the heading line.

For example:

a) .hm 4

b) +2

c) +6 blank lines (1 inch)

= .tm 12

This control word acts as a BREAK.

Example:

.TM 3

Three lines will be left at the top of pages started after the current page. The heading and page number will be printed on the first line and the subheading on the second line under the default heading margin.

UNDENT Control

Purpose:

The UNDENT control word forces the immediately following line to start further left than the position indicated by the current indent.

Format:

.UN <n>

<n> specifies the number of spaces to be "undented" (negative indent) for the next line only; it must be less than or equal to the amount of indent currently in effect.

Usage:

The UNDENT control word serves the same purpose as the OFFSET control word but in a different manner. It is usually used to make the first line of a paragraph or section extend further to the left than the body of the paragraph. The choice between using UNDENT and OFFSET is usually a matter of personal preference. In general, UNDENT is more convenient once one becomes familiar with its usage.

Note:

This control acts as a BREAK.

Examples:

.UN 10

If an indentation of 10 is in effect, the next line will start at the left margin; all following lines will occur at normal indent position, 10 spaces from the left margin.

Appendix I
Text Output Processor Operation

The TDL Text Output Processor requires 4K of memory, a TDL system monitor (either ZAP, ZAPPLE, or SMB), a reader device and a list device for proper operation. The reader MUST be under software control.

After loading the Output Processor by using the R command of the monitor, the previously prepared source file is readied in the reader device. The Processor is started by issuing the G command to the monitor. The Output Processor will sign on on the console, and then wait. At this time, adjust the paper in the output device so that printing will begin on the first line of the page. A CR (carriage return) is then entered to start the output.

Upon completion of the output, the Processor will TRAP to the monitor. If additional source is to be processed, the Processor may be restarted by simply issuing another G command to the monitor.

During operation, the output may be temporarily stopped by entering a Control-S on the console (provided it is not being used as the reader device). A Control-Q is entered to continue output. A Control-C will abort the Processor and TRAP back to the monitor. To resume output, use the monitor's G command.

In a constant effort to improve the quality and usefulness of its publications, Technical Design Labs, Inc. provides this page for user feedback. Your critical evaluations of this document is our only effective means of determining its serviceability. Please give specific page and line references where applicable.

ERRORS NOTED IN THIS PUBLICATION:

SUGGESTIONS FOR IMPROVING THIS PUBLICATION: (i.e. clarity, organization,convenience,accuracy,legibility.)

MISSING DOCUMENTATION: (i.e. completeness.)

Name----------Date----------
Street----------
City----------State----------Zip Code----------

All comments and suggestions become the property of TDL. Send to Technical Design Labs, Inc.
Dept. of Product Improvements
1101 State Road
Princeton, N. J. 08540

Please indicate in the space below if you wish a reply.

Z-80 RELOCATING/LINKING

ASSEMBLER

USER'S MANUAL

Kitan, Inc.
Research Park, Bldg. H.
1101 State Road
Princeton, N.J. 08540

TDL Z80 Relocating/Linking Assembler

User's Manual

Revision 2.2

October 15, 1977

Written by Neil J. Colvin

Chapter 1

Introduction

The TDL Z80 Relocating/Linking Assembler is the symbolic assembly program for the Z80. It is a two-pass assembler (requiring the source program to be read twice to complete the assembly process) designed to run under the TDL system monitor. It is therefore device independent, allowing complete user flexibility in the selection of standard input and output device options.

The assembler performs many functions, making machine language programming easier, faster, and more efficient. Basically, the assembler processes the Z80 programmer's source program statements by translating mnemonic operation codes to the binary codes needed in machine instructions, relating symbols to numeric values, assigning relocatable or absolute memory addresses for program instructions and data, and preparing an output listing of the program which includes any errors encountered during the assembly.

The TDL Z80 Assembler also contains a powerful macro capability which allows the programmer to create new language elements, thus expanding and adapting the assembler to perform specialized functions for each programming job.

In addition, the TDL Assembler provides the facilities required to specify program module linkages, allowing the TDL Linkage Editor to link independently assembled program modules together into a single executable program. This allows for the modular and systematic development of large programs, and for easy sharing of common program modules among different programs.

Statements

Assembler programs are usually prepared on a terminal, with the aid of a text editing program. A program consists of a sequence of statements in the assembly language. Each statement is normally written on one line, and terminated by a carriage return/line feed sequence. TDL assembler statements are free-format. This means that the various statement elements are not placed at a specific numbered column on the line.

There are four elements in an assembler statement (three of which are optional), separated from each other by specific characters. These elements are identified by their order of appearance in the statement, and by the separating (or delimiting) character which follows or precedes the elements.

Statements are written in the general form:

```
label: operator   operand,operand ;comment <CR-LF>
```

The assembler converts statements written in this form into the binary machine instructions.

Instruction Formats

The Z80 uses a variable length instruction format. A given machine instruction may be one, two, three, or four bytes long depending on the specific machine code and on the addressing mode specified. The TDL assembler automatically produces the correct number of machine code bytes for the particular operation specified. Appendix A specifies the various machine code mnemonics accepted by the assembler and the format of the operands required.

Statement Format

As previously described, assembler statements consist of a combination of a label, an operator, one or more operands, and a comment; the particular combination depends on the statement usage and operator requirements.

The assembler interprets and processes these statements, generating one or more binary instructions or data bytes, or performing some assembly control process. A statement must contain at least one of these elements, and may contain all four. Some statements have no operands, while others may have many.

Statement labels, operators, and operands may be represented numerically or symbolically. The assembler interprets all symbols and replaces them with a numeric (binary) value.

Symbols

The programmer may create symbols to use as statement labels, as operators, and as operands. A symbol may consist of any combination of from one to six characters from the following set:

The 26 letters: A-Z
Ten digits: 0-9
Three special characters:
- $ (Dollar Sign)
- % (Percent)
- . (Period)

These characters constitute the Radix-40 character set (so named because it contains only 40 characters). Any statement character which is not in the Radix-40 set is

treated as a symbol delimiter when encountered by the assembler.

The first character of a symbol must not be numeric. Symbols may also not contain embedded spaces. A symbol may contain more than six characters, but only the first six are used by the assembler.

The TDL assembler will accept programs written using both upper and lower case letters and symbols. Lower case letters are treated as upper case in symbols. Additional special characters and lower case letters elsewhere are taken unchanged.

Labels

A label is the symbolic name created by the programmer to identify a statement. If present, the label is written as the first item in a statement, and is terminated by a colon (:). A statement may contain more than one label, in which case all identify the same statement. Each label must be followed by a colon, however. A statement may consist of just a label (or labels), in which case the label(s) identifies the following statement.

When a symbol is used as a label, it specifies a symbolic address. Such symbols are said to be defined (have a value). A defined symbol can reference an instruction or data byte at any point in the program.

A label can be defined with only one value. If an attempt is made to redefine a label with a different value, the second value is ignored, and an error is indicated.

The following are legal labels:

```
$SUM:
ABC:
B123:
WHERE%:
```

The following are illegal:

```
30QRT:      (First character must not be a digit)
AB CD:      (Cannot contain embedded space)
```

If too many characters are used in a label, only the first six are used. For example the label ZYXWVUTSR: is recognized by the assembler to be the same as ZYXWVUABC:.

Operators

An operator may be one of the mnemonic machine instruction codes, a pseudo-operation code which directs the assembly process, or a user defined code (either pseudo-op or macro). The assembler pseudo-op codes are described in Chapter 3 and summarized in Appendix B.

The operator element of a statement is terminated by any character not in the Radix-40 set (usually a space or a tab). If a statement has no label, the operator must appear first in the statement.

A symbol used as an operator must be predefined by the assembler or the programmer before its first appearance as an operator in a statement.

Operands

Operands are usually the symbolic addresses of the data to be accessed when an instruction is executed, the names of processor registers to be used in the operation, or the input data or arguments to a pseudo-op or macro instruction. In each case, the precise interpretation of the operand(s) is dependent on the specific statement operator being processed. Operands are separated by commas, and are terminated by a semicolon (;) or a carriage return/line feed.

Symbols used as operands must have a value predefined by the assembler or defined by the programmer. These may be symbolic references to previously defined labels where the arguments used by this instruction are to be found, or the symbols may represent constant values or character strings.

Comments

The programmer may add a comment to a statement by preceding it with a semicolon (;). Comments are ignored by the assembler but are useful for documentation and later program debugging. The comment is terminated by the carriage return/line feed at the end of the statement. In certain cases (e.g. conditional assembly and macro definitions), the use of the left and right square brackets ([]) should be avoided in a comment as it could affect the assembly process.

An assembler statement may consist of just a comment, but each such statement must begin with a semicolon.

Statement Processing

The assembler maintains several internal symbol tables for recording the names and values of symbols used during the assembly. These tables are:

1. Macro Table - This table contains all macros. It is initially empty, and grows as the programmer defines macros.

2. Op-Code Table - This table contains all of the machine operation mnemonics (op-codes), the assembler pseudo-operations (pseudo-ops), and user defined

operators (.OPSYNs). It initially contains the basic op-codes and pseudo-ops, and grows as the programmer provides additional definitions.

3. Symbol Table - This table contains all programmer-defined symbols other than those described above. It initially contains the standard register names, and then grows as new symbols are defined.

Internally, all of these tables occupy the same space, so that all of the available space can be used as required.

Order of Symbol Evaluation

The following table shows the order in which the assembler searches the tables for a symbol appearing in each of the statement fields:

Label Field:
1. Symbol followed by a colon. If no colon is found, no label is present.

Operator Field:
1. Macro
2. Machine operator
3. Assembler operator
4. Symbol

Operand Field:
1. Number
2. Macro
3. Symbol
4. Machine operator

Because of the different table searching orders for each field, the same symbol could be used as a label, an operator, and a macro, with no ambiguity.

Programmer-Defined Symbols

There are two types of programmer-defined symbols: labels and assignments. As previously described, labels are generated by entering a symbol followed by a colon (e.g. LABEL:). Symbols used as labels cannot be redefined with a different value once they have been defined. The value of a label is the value of the location counter at the time the label is defined.

Assignments are used to represent, symbolically, numbers, bit patterns, or character strings. Assignments simplify the program development task by allowing a single source program modification (the assignment statement) to change all uses of that number or bit pattern throughout the

program. Symbols given values in an assignment statement may have new values assigned in subsequent statements. The current value of an assigned symbol is the last one given to it.

A symbol may be entered into the symbol table with its assigned value by using a direct assignment statement of the form:

```
symbol = value {; or CR-LF}
```

where the value may be any valid numeric value or expression.

The value assigned to the symbol may subsequently be changed by another direct assignment statement.

The following are valid assignment statements:

```
VALUE1 = 23
SIZE = 4*36
ZETA = SIZE
```

If it is desired to fix the value assigned to a symbol so that it cannot subsequently be redefined, the direct fixed assignment statement should be used. This statement is the same as the direct assignment statement except that the symbol is followed by two equal signs instead of one. For example:

```
FIXED == 46
NEWVAL == SIZE
```

Assembly-Time Assignments

It is often desirable to defer the assignment of a value to a symbol until the assembly is actually underway (i.e. not specify the value as part of the source program). This is especially useful in setting program origin, buffer sizes, and in specifying parameter values which will be used to control conditional assembly pseudo-ops.

The TDL Assembler provides the ability to specify symbols with values to be determined at assembly time, and the mechanism by which the values may be interactively defined. To specify an assembly-time assignment, the following format is used:

```
symbol =\ [dtextd]
```

where the dtextd in brackets indicates the optional specification of a message to be output on the console device at assembly time before requesting the symbol's value. The d represents a text delimiter, and may be any character (other than a space or tab) which is not contained in the text itself. The text may contain carriage

return/line feed sequences, which would result in a multi-line message on the console.

After the optional message is output on the console, a colon (:) is output to indicate that the assembler is waiting for the desired value to be entered. The value which is to be assigned to the symbol is then input on the console device and the assembly continues with the symbol having the specified value. This interaction only occurs during the first assembly pass. The symbol's value remains unchanged during subsequent passes.

Only numeric values may be entered through the console in this fashion. The number which is input must conform to the same rules as any other number used in the assembly source program, and may be followed by an optional radix modifier (see the section on Numbers below). The number is assumed to be decimal unless followed by a radix modifier.

The value being input is not processed until a carriage return is entered. Any mistyped character may be deleted by the use of the DELETE (or RUBOUT) key (which will echo the deleted character), and the entire number may be deleted by entering CTL-U (simultaneous use of the CTRL and the U key). Any character which is input but is not valid as part of a number will not be echoed and will be ignored.

The following are examples of assembly-time assignment statements:

```
BUFSIZ =\ "BUFFER SIZE (50 TO 500 CHARACTERS)"
DISK =\ "VERSION (0-PAPER TAPE 1-DISK)"
```

Assembly-time assignment statements are similar to direct fixed assignments (==) in not allowing the symbol to be redefined elsewhere in the program.

Local and Global Symbols

When assembling a large program, it is sometimes difficult to keep track of the symbols used for local data references and branching. To facilitate modular programming, the TDL assembler provides for both global and local symbols within a single program. All symbols which start with two periods are defined as being local, and all other symbols are global. For example, the following are valid local symbols:

```
..ABCD:
..1234:
..:
```

A particular occurrence of a local symbol is only defined within the boundaries of its enclosing global symbols. For example, in the following sequence of label definitions, the symbol ..SYM1 is only defined (and can only be referenced)

within the program between the definition of GLOB1 and GLOB2:

```
      ...
GLOB1:
      ...
..SYM1:
      ...
GLOB2:
      ...
```

This localization of symbol definitions allows the same symbol to be used unambiguously more than once in the program. It also simplifies program understandability by immediately differentiating between local and global symbols.

In addition to labels, any other programmer-defined symbol may be specified as local (e.g. macros) in the same manner. Because of the local usage of these symbols, they do not appear in the symbol table listing or in the symbol table optionally punched on the object tape.

External, Internal, and Entry Symbols

Programmer-defined symbols may also be used as external, internal, and entry point symbols in addition to their appearance as labels or in assignment statements.

Symbols which fall into one of these three groups are different from other symbols in the program because they can be referenced by other, separately assembled, program modules. The manner in which they are used depends on where they are located: in the program in which they are defined, or in the program in which they are a reference to a symbol defined elsewhere.

If the symbol appears in a program in which it is defined, it must be declared as being available to other programs by the use of the pseudo-ops .INTERN or .ENTRY, or through the use of the delimiters "::", "=:", "==:", or "=\:" in their definition statements. These special delimiters are exactly equivalent to the sequence:

```
.INTERN symbol
symbol <delimiter without colon (:)>
```

In each case, the delimiter is the normal symbol definition operator (:, =, ==, =\) with an additional colon (:) added to indicate an internal symbol definition.

If the symbol is located in a program in which it is a reference to a symbol defined in another program, it must be declared as external by the use of the .EXTERN pseudo-op, or through the use of the "#" symbol modifier. This special symbol modifier is appended to the end of any symbol to

declare it external. For example, the statement:

```
LXI H,SYMBOL#
```

is exactly equivalent to:

```
.EXTERNAL SYMBOL
LXI H,SYMBOL
```

Numbers

Numbers used in a program are interpreted by the assembler according to a radix (number base) specified by the programmer, where the radix may be 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). The programmer uses the .RADIX pseudo-op to set the radix for all numbers which follow. If the .RADIX statement is not used, the assembler assumes a radix of 10 (decimal).

The radix may be changed for a single number by appending a radix modifier to the end of the number. These modifiers are B for binary, O or Q for octal, D or . (period) for decimal, and H for hexadecimal. To specify the hexadecimal digits, the letters A through F are used for the values 10 through 15 decimal. All numbers, however, must begin with a numeral. For example, the following are valid numbers:

```
10        10 in current radix
10.       10 decimal
10B       10 binary (2 decimal)
0FFH      FF hexadecimal (255 decimal)
```

The following are invalid numbers:

```
14B       4 is not a binary digit
FFH       the number must start with a numeral
```

Arithmetic and Logical Operations

Numbers and defined symbols may be combined using arithmetic and logical operators. The following operators are available:

```
+   Add (or unary plus)
-   Subtract (or unary minus)
*   Multiply
/   Integer division (remainder discarded)
@   Integer remainder (quotient discarded)
&   Logical AND
!   Logical inclusive OR
^   Logical exclusive OR (or unary radix change)
#   Logical unary NOT
```

< Left binary shift
> Right binary shift

The assembler computes the 16-bit value of a series of numbers and defined symbols connected by these operators. All results are truncated to the left, if necessary. Two's complement arithmetic is used, with the meaning of the sign bit (the most significant bit) being left to the programmer. This means that a numeric value may be either between 0 and 65,535 or between -32,768 and 32,767, depending on whether it is signed or unsigned.

These combinations of number and defined symbols with arithmetic and logical operators are called expressions. When evaluating an expression, the assembler performs the specified operations in a particular order, as follows:

1. Unary minus or plus (- +)
2. Unary radix change (^B ^O ^Q ^D ^H)
3. Left and right binary shift (< >)
4. Logical operators (& ! ^ #)
5. Multiply/Divide (* /)
6. Remainder (@)
7. Add/Subtract (+ -)

Within each of the above groups, the operations are performed from left to right. For example, in the expression:

```
-ALPHA+3*BETA/DELTA&^H55
```

the unary minus of ALPHA is done first, then DELTA is ANDed with a hexadecimal 55, then BETA is multiplied by 3, the result of which is divided by the result of the AND, and finally, that result is added to the negated ALPHA.

To change the order in which the operations are performed, parentheses may be used to delimit expressions and to specify the desired order of computation. Each expression within parentheses is considered to be a single numeric value, and is completely evaluated before it is used to compute any further values. For example, in the expression:

```
4*(ALPHA+BETA)
```

the addition of ALPHA to BETA is peformed before the multiplication.

Radix Change Operator

The radix change operator is used to temporarily change the radix in which a following number or expression is to be interpreted. It is written as an up-arrow (^) followed by

the radix modifier of the desired radix. These modifiers are the same as those used to specify the radix of a single number (B-binary, O or Q-octal, D-decimal, and H-hexadecimal). The radix change only affects the immediately following number or parenthesized numeric expression. For example, all of the following are valid representations of the decimal number 33:

```
33.
33D
^D33
^D(10*3+3)
^D(10*THREE+THREE)
^D10*^D3+^D3
```

but the following is not a representation of decimal 33 if the prevailing radix is not decimal:

```
^D3*10+3
```

because the radix change only affects the value immediately following it, in this case 3.

Binary Shifting

The binary shift operators (< left, > right) are used to logically shift a 16-bit value to the left or right. The number of places to be shifted is specified by the value following the shift operator. If that value is negative, the direction of the shift is reversed. For example, all of the following expressions have a value of 4 decimal:

```
8>1
1<2
2>-1
```

One-byte Values

All of the above discussion has been based on the computation of 16-bit (two byte) numeric values. Many of the Z80 operations require an 8-bit (one byte) value. Since all computations are done as a 16-bit value, an operation calling for only eight bits will discard the high order eight bits (the most significant byte) of the value. If the byte discarded is not either zero or minus one (all one bits), a warning will be given on the assembly listing.

Character Values

To generate a binary value equivalent to the ASCII representation of a character string, the single (') or double (") quotation mark is used. The character string is

enclosed in a pair of the quotation marks. For example, all of the following are valid character values:

```
"A"
'B'
"AB"
'CD'
```

Note that whichever quotation mark is used to initiate the character string it must also be used to terminate it. If the string is longer than two bytes, it is truncated to the left. Each 7-bit ASCII character is stored in an 8-bit byte, with the high-order bit set to zero.

A character string of this type may be used wherever a numeric value is allowed.

A single quote may be used inside a string delimited by double quotes, and vice-versa. If it is necessary to use a single quote within a string delimited by single quotes, two single quotes must be used. The same is true for a double quote in a string delimited by double quotes.

Location Counter Reference

The location counter may be referenced as a numeric 16-bit value by the use of the symbol . (period). The value represented by . is always the location counter value at the start of the current assembly language statement. For example:

```
JMP .
```

is an effective error trap, jumping to itself continuously.

Chapter 2

Addressing and Relocation

Address Assignment

As source statements are processed by the assembler, consecutive memory addresses are assigned to the instruction and data bytes of the object program. This is done by incrementing an internal program counter each time a memory byte is assigned. Some statements may increment this internal counter by only one, while others could increase it by a large amount. Certain pseudo-ops and direct assignment statements have no effect on the counter at all.

In the program listing generated by the assembler, the address assigned to every statement is shown.

Relocation

The TDL Z80 Assembler will create a relocatable object program. This program may be loaded into any part of memory as a function of what has been previously loaded. To accomplish this, certain 16-bit values which represent addresses within the program must have a relocation constant added to them. This relocation constant, added when the program is loaded into memory, is the difference between the memory location an instruction (or piece of data) is actually loaded into, and the location it was assembled at. If an instruction had been assembled at location 100 (decimal), and was loaded into location 1100 (decimal), then the relocation constant would be 1000 (decimal).

Not all 16-bit quantities must be modified by the relocation constant. For example, the instruction:

```
LXI H,00FFH
```

references a 16-bit quantity (00FFH) which does not need relocation. However, the set of instructions:

```
JZ DONE
  .
  .
  .
DONE:
```

does reference a 16-bit quantity (the address of DONE) which must be relocated, since the physical location of DONE changes depending on where the program is loaded into memory.

To accomplish this relocation, the 16-bit value forming

an address reference is marked by the assembler for later modification by the loader or linkage editor. Whether a particular 16-bit value is so marked depends on the evaluation of the arithmetic expression from which it is obtained. A constant value (integer) is absolute (not relocatable), and never modified. Point references (.) are relocatable (assuming relocatable code is being generated), and are always modified by the loader or linkage editor. Symbolic references may be either absolute or relocatable.

If a symbol is defined by a direct assignment statement, it may be absolute or relocatable depending on the expression following the equal sign (=). If the symbol is a label (and relocatable code is being generated) then it is relocatable.

To evaluate the relocatability of an expression, consider what happens at load or linkage edit time. A relocation constant, r, must be added to each relocatable element, and the expression evaluated. For example, in the expression:

```
Z = Y+2*X-3*W+V
```

where V, W, X, and Y are relocatable. Assume that r is the relocation constant. Adding this constant to each relocatable term, the expression becomes:

```
Z(r) = (Y+r)+2*(X+r)-3*(W+r)+(V+r)
```

By rearranging the expression, the following is obtained:

```
Z(r) = Y+2*X-3*W+V + r
```

This expression is suitable for relocation because it contains only a single addition of the relocation constant r. In general, if the expression can be rearranged to result in the addition of either of the following, it is legal:

```
0*r     absolute expression
1*r     relocatable expression
```

If the rearrangement results in the following, it is illegal:

```
n*r     where n is not 0 or 1
```

Also, if the expression involves r to any power other than 1, it is illegal. This leads to the following rules:

1. Only two values of relocatability for a complete expression are allowed (ie. n*r where n = 0 or 1).
2. Division by a relocatable value is illegal.

3. Two relocatable values may not be multiplied together.
4. Relocatable values may not be combined by logical operators.
5. A relocatable value may not be logically shifted.

If any of these rules is broken, the expression is illegal and an error message is given.
If X, Y, and Z are relocatable symbols, then:

```
X+Y-Z     is relocatable
X-Z       is absolute
X+7       is relocatable
3*X-Y-Z   is relocatable
4&X-Z     is illegal
```

Only 16-bit quantities may be relocated. All 8-bit values must be absolute or an error will be given.

Relocation Bases

One of the unique capabilities of the TDL Z80 Assembler is its ability to handle symbolic references to separately located areas of memory, where the mapping of symbols to physical addresses occurs at linkage edit time. The symbolic names for independently located memory areas are called "relocation bases". These relocation bases may represent ROM vs. RAM, shared COMMON areas, special memory areas such as video refresh, memory mapped I/O, etc. Within each subprogram, each of these memory areas is referenced by a unique name, with the actual allocation deferred to the link edit and load process. All memory references within the assembled program are relative to one of these relocation bases.

As each relocation base is assigned a name in the program (through the use of the .EXTERN pseudo-op), it is implicitly assigned a sequential identifying number. This number appears in the listing as part of any address relative to that base.

Four of these relocation bases (0-3) have predefined names and meanings, and are treated differently at linkage edit time than the remainder of the bases. Base 0 represents absolute memory locations (i.e. it always has the value of 0). Base 1 has the name .PROG. and represents the program area (maybe PROM or ROM). Most program code (and data in non-rommed programs) is generated relative to this relocation base. Base 2 has the name .DATA. and represents the local data area for each module. Most local data is defined relative to this base. Base 3 has the name .BLNK. and represents the global "blank common". This relocation base is always assigned the value of the first free byte in memory after the local data storage (.DATA.) and other data relocation segments by the linkage editor. Because it is

always the last allocated, modules referencing this area can be included in any order, regardless of the amount of the area they use.

Relocation segments relative to bases 1 and 2 (.PROG. and .DATA.) are always allocated additively (i.e. after each module is allocated, the value of the relocation base is increased by the size of the segment). All other relocation bases are normally assumed to have constant values during the allocation process (usually assigned by the linkage editor).

Each symbol defined during the assembly has a relocation base associated with it. There are no limitations on inter-base references (i.e. code relative to .PROG. can freely reference data relative to .DATA.). Expressions containing symbols must evaluate to a value relative to a single relocation base, but may contain references to multiple relocation bases. All relocation base references except for the final result must be part of sub-expressions which evaluate to absolute values. For example, if T and U are symbols relative to base 1, V and W relative to base 2, and X and Y relative to base 3, then the following are valid expressions:

```
T+(V-W)         (note the parentheses to make V-W
                 a subexpression)
X+3
T-(V-W)*U+(X-Y)
```

and the following are invalid:

```
T+U             (within a relocation base, the
                 normal relocation rules apply)
T+V-W           (T+V is the first subexpression,
                 and it is mixed relocation bases)
```

It should be noted that conceptually, normal external symbols are simply relocation bases with a size of zero (0), and the assembler treats them that way. An assignment of the form:

```
N==P+5
```

where P is an external symbol, makes N a symbol whose address is relative to P, even though P has no size. Hence, expressions of the form:

```
5*(P-N)
```

where P and N have the same relocation base, are in fact valid.

Chapter 3

Pseudo-Operations

Pseudo-operations (pseudo-ops) are directions to the assembler to perform certain operations for the programmer, as opposed to machine operations which are instructions to the computer. Pseudo-ops perform such functions as listing control, data conversion, or storage allocation.

Address Mode and Origin

The TDL Z80 Assembler normally assembles programs in relocatable mode, so that the resultant program can be loaded anywhere in memory for execution. Therefore, all programs are assembled assuming their first byte is at address zero (0), because they can be relocated anywhere. When desired, however, the assembler will generate absolute object code, either for the entire program, or just selected portions. The assembler will also locate the assembled code at any address desired. The two pseudo-ops which control address mode, relocation base and address origin are .LOC and .RELOC.

.LOC n

This statement sets the location counter to the value n, which may be any valid expression. If n is an absolute value, then the assembler will assign absolute addresses to all of the instructions and data which follow. If n is relocatable, then relocatable addresses will be assigned, relative to the relocation base of the expression.

The program is assumed to start with an implicit .LOC to relocatable address zero (0) of the relocation base named .PROG. (the default relocation base for normal programs). A program can contain more than one .LOC, each controls the assignment of addresses to the statements following it.

To reset the program counter to its value prior to the last .LOC, the statement:

.RELOC

is used. This statement restores both the value, the relocation base and the addressing mode which were in effect before the immediately preceding .LOC. If no .LOC has been done, then a .RELOC is equivalent to a:

.LOC 0

When in relocatable addressing mode, the assembler will determine whether each 16-bit value is absolute or relocatable as described in Chapter 2.

Data Definition

The TDL Z80 Assembler provides a number of different pseudo-ops for describing and entering data to be used by the program.

.RADIX

When the assembler encounters a number in a statement, it converts it to a 16-bit binary value according to the radix indicated by the programmer. The statement:

```
.RADIX n
```

where n is 2, 8, 10, or 16, sets the radix to n for all numbers which follow, unless another .RADIX statement is encountered, or the radix is modified by the ^r operator or a suffix radix modifier.

The statement:

```
.RADIX 10
```

implicitly begins each assembly program, setting the initial radix to decimal.

.BYTE

To enter one (or more) 8-bit (one byte) data values into the program, the statement:

```
.BYTE n {, n ...}
```

where n is any expression with a valid 8-bit value is used. More than one byte can be defined at a time by separating it from the preceding value with a comma. All of the bytes defined in a single .BYTE statement are assigned consecutive memory locations. For example:

```
.BYTE 23,4*^H0FF,BETA-ALPHA
```

defines three sequential bytes of data.

.WORD

To enter a 16-bit (two byte) value into the program, the statement:

```
.WORD nn {, nn ...}
```

where nn is any expression with a valid 16-bit value, is used. Multiple 16-bit values may be defined with one .WORD statement by separating each from the preceding one with a comma.

All 16-bit values defined by the .WORD pseudo-op are stored in standard Z80 word format, least significant byte first.

For example, the following statement:

```
.WORD ALPHA,234*BETA,^H0EEFF
```

defines three sequential 16-bit values, or a total of six bytes of data.

.ASCII, .ASCIZ, and .ASCIS

To enter strings of text characters into the program, one of the statements:

```
.ASCII dtextd | [n]
.ASCIZ dtextd | [n]
.ASCIS dtextd | [n]
```

is used. The d represents a text delimiter, and may be any character (other than space or tab) not contained in the text itself. Each character in the text is converted to its 7-bit ASCII representation (with the eighth bit zero), and stored in sequential memory locations. When the delimiter character is again encountered, the text is considered terminated (the delimiter is not stored with the string). The delimited string may be followed by another delimiter, and another string, and this may be repeated as desired.

If it is necessary to include values in the text string for which no character exists, then the second option shown above may be used. If in place of a string delimiter, the assembler finds a left square bracket ([), then the numeric expression enclosed within it and a matching right square bracket (]) is evaluated as an 8-bit value and stored as the next byte of the string. These 8-bit values may be intermixed with delimited strings as required.

It is important to note that tab, carriage return, and line feed are all valid characters within a delimited text string. It is therefore possible that a .ASCIx statement will encompass more than one line in the source program.

The difference between the three pseudo-ops described above is in their treatment of the last byte generated by the statement. The .ASCII statement just stores the byte. The .ASCIZ statement stores one additional byte after the last one, a null (zero) byte to mark the end of the string in memory. The .ASCIS

pseudo-op sets the high-order (eighth) bit of the last byte to one to flag the last byte.

The following are all valid .ASCIx statements:

```
.ASCII /This is a string/
.ASCIZ /This is two/ ' strings in one place'
.ASCIS [^H0D] [^H0A] "Message on new line"
.ASCII \
Message on new line\
```

.RAD40

The Radix-40 character set for symbols was chosen because it allows a six character symbol to be stored in only four bytes of memory. To allow the program to define data bytes in this character set, the statement:

```
.RAD40 symbol1 {, symbol2 ...}
```

is used. The symbol must conform to all the rules specified for assembler symbols, and is converted into the Radix-40 notation and stored in four sequential bytes of memory. If multiple symbols are to be converted and stored, each must be separated from the preceding one by a comma.

Storage Allocation

The TDL Z80 Assembler allows the programmer to reserve single locations, or blocks of many locations, for use during the execution of the program. The two pseudo-ops used for this purpose are .BLKB and .BLKW. The format of the statement using these pseudo-ops is:

```
.BLKx n
```

where n is the number of storage locations to be reserved.

For the .BLKB pseudo-op, each storage location consists of one byte, so the above statement will reserve n contiguous bytes of memory, starting at the current location counter. The .BLKW pseudo-op uses a word (two bytes) as its storage unit, so the above statement would reserve n words, or two times n bytes of contiguous memory.

For example, each of the following statements reserves 24 (decimal) bytes of storage:

```
.BLKB 24.
.BLKW ^D12
.BLKB 2*12.
```

Program Termination

Every program must be terminated by a .END pseudo-op. The format of this statement is:

```
.END start
```

where start is an optional starting address for the program. The starting address is normally only necessary for the main program. Subprograms, which are called from the main program, need no starting address.

When the assembler encounters the .END pseudo-op during pass 1 of the assembly, it returns to the initialization point to await further instructions (see Appendix C). On a listing pass, the .END pseudo-op initiates the printing of the symbol table (if not suppressed by a prior .XSYM pseudo-op). On a punching pass, the .END pseudo-op punches the EOF record on the object tape.

Subprogram Linkage

Programs usually consist of a main program and numerous subroutines which communicate with each other through parameter linkages and through reference to symbols defined elsewhere in the program. Since the TDL Z80 Assembler provides the means for the various program components to be assembled separately from each other, the linkage editor (which finally puts the pieces together) must be able to identify those symbols which are references (or referenced) external to the current program. For a given subprogram, these "linkage" symbols are either symbols defined internally which must be available to other programs to reference, or symbols used internally but defined externally to the program. Symbols defined within the program but available to other subprograms are called "internal" symbols. Symbols used internally but defined elsewhere are called "external" symbols.

To set up these linkages between subprograms, four pseudo-ops are provided: .IDENT, .EXTERN, .INTERN, and .ENTRY.

The .IDENT statement has the format:

```
.IDENT symbol
```

where symbol is the relocatable module name. This name is used by the linkage editor to identify the module on memory allocation maps, and to allow the selective loading of the module if it is part of a subprogram library. If the .IDENT statement does not appear in a program, the name ".MAIN." is assumed. The .IDENT name appears at the top of every listing page, and is displayed on the console at the start of the second assembly pass of that module.

All three remaining statements have the same format:

```
.EXTERN symbol1 {, symbol2 ...}
.INTERN symbol1 {, symbol2 ...}
.ENTRY symbol1 {, symbol2 ...}
```

where symbol1 is the symbol being declared as external, internal, or as an entry point. Multiple symbols may be declared in the same statement by separating each from the preceding one with a comma.

The .EXTERN statement identifies symbols which are defined elsewhere. External symbols must not be defined within the current subprogram. The external symbols may only be used as addresses, or in expressions that are to be used as addresses. External symbols may be used in the same manner as any other relocatable symbol, with the following restrictions:

1. The use of more than one external symbol in a single expression is illegal. Thus X+Y where X and Y are both external is illegal.
2. Externals may only be additive. Therefore the following expressions are illegal (where X is an external symbol):

```
-X
2*X
SQR-X
2*X-X
```

Symbols declared as external by the .EXTERN pseudo-op may also be used as relocation bases. This is done by using an external symbol as the argument to a .LOC pseudo-op. All memory allocated by the assembler after the .LOC will be addressed relative to the specified relocation base. The most common use of this capability is the declaration of COMMON blocks for the sharing of data between assembler and FORTRAN subprograms. All named COMMON blocks are in fact just different relocation bases. Symbols used as relocation bases have unique values during the assembly of the program module. At any point in time, the current value of the relocation base symbol is the number of bytes which have been allocated to that base so far. This means that subsequent .LOC pseudo-ops referencing the same external symbol will start the memory allocation at the next available byte in that relocation base, not at relative location zero (0).

There are three predefined relocation base symbols: .PROG., .DATA. and .BLNK.. These relocation bases are used for the program code, separately located data (in a ROM/RAM environment), and blank (unnamed) common respectively.

The .INTERN pseudo-op identifies those symbols within the current subprogram which are to be made accessible to

other programs as external symbols. This statement has no effect on the assembly process for the current program, but merely records the name and value of the identified symbols on the object tape for later use by the linkage editor. An internal symbol must be defined within the current program as a label, or in a direct assignment statement.

The .ENTRY pseudo-op functions identically to the .INTERN pseudo-op, with one addition. It is sometimes desirable to put many subroutines with common usage into one "library", and to allow the linkage editor to select only those programs from the library which are called by the program being linkage edited.

The .ENTRY statement, in addition to functioning as a .INTERN statement, also flags the specified symbols as program entry points. If the subprogram is later put into a library, this will specify to the linkage editor that this program is to be included only if one of its entry points is referenced as an external symbol by an already included program.

Since these entry points are external to the program referencing them, they must be listed in a .EXTERN statement in the calling program.

Listing Control

Program listings are printed on the list device during pass 2 and 4 (see Appendix C) of the assembly. The listing is printed as the source program statements are processed during the pass. The standard listing contains (from left to right):

1. Error flags (if present).
2. Location counter for the first byte generated by this statement.
3. Instruction or data in hexadecimal (maximum of five bytes per line printed).
4. Exact image of the input statement.

The standard listing displays all 16-bit quantities in 16-bit (two byte), most significant byte first, format. These quantities are properly reversed in the object code as required by the Z80. A 16-bit relocatable address relative to the .PROG. relocation base is flagged with an apostrophe ('), one relative to the .DATA. relocation base is flagged with an asterisk (*), and all others are followed by the assigned number of their relocation base.

Within a macro expansion, only the macro call and those statements which generate actual object code are normally listed.

If a single statement generates more than the maximum of five bytes that can be listed on a single line, the remaining bytes are properly generated, but not normally

listed.

A listing always begins at the top line of the page, and 60 lines are printed per page, with a two line margin at the top, and a two line margin at the bottom. A page is assumed to be 72 (or 79) columns wide (depending on the list device selected - see Appendix C). Each page is numbered, and can have an optional title and sub-title.

The standard listing options can be changed and expanded by the use of the following pseudo-operations:

.PAGE This statement causes the assembler to skip to the top of the next page (by counting lines). A form feed character in the input text will have the same effect.

.XLIST This statement causes the assembler to stop listing the assembled program at this point.

.LIST This statement is normally used following a .XLIST to resume program listing.

.LALL This statement causes the assembler to list everything which is processed. This includes all text, macro expansions, and all other statements normally suppressed in the standard listing.

.XALL This statement is normally used following a .LALL to resume the normal listing.

.SALL This statement causes the suppression of all macro expansions and their text. It can be reset by a subsequent .LALL or .XALL.

.XSYM This statement suppresses the symbol table listing normally performed upon encountering the .END statement.

.LSYM Normally not used, this statement enables the listing of the symbol table previously suppressed by the .XSYM pseudo-op.

.LADDR This statement causes the assembler to list all 16-bit quantities in the same order it generates them in the object code (least significant byte first).

.XADDR Normally used following a .LADDR statement, this statement resumes the normal listing of 16-bit quantities in non-swapped format.

.LIMAGE This statement causes the assembler to list every byte generated, even if more than one line (at five bytes per line) is required. In this mode, the assembler will attempt to split the input source statement to indicate which part of the statement is generating which bytes.

.XIMAGE Normally used following a .LIMAGE statement, this statement resumes the normal listing of only five bytes of generated data per statement.

.LCTL This statement causes all subsequent listing control statements (e.g. .XLIST) to be listed themselves. Normally, no listing control statement is itself listed. The .XCTL pseudo-op is used to reset this option.

.XCTL Normally used following a .LCTL statement, this statement resumes the default suppression of the listing of listing control statements.

.SLIST This statement causes the current listing control flags to be saved on a four element push-down stack. The current flag settings remain unchanged. These settings may later be restored with the .RLIST pseudo-op. This pseudo-op may be followed on the same line with another listing control pseudo-op, which will take effect prior to the listing of the .SLIST statement.

.RLIST This statement restores the listing control flags from the top element of the .SLIST push-down stack. These new flags take effect with the statement following the .RLIST.

.TITLE dtextd This statement defines the delimited string text to be the title to be printed at the top of every page of the listing. The text must be delimited in the same manner as in the .ASCII pseudo-op, and must be no longer than 72 characters. If the .TITLE pseudo op is the first statement on a page, then the new title will be printed at the top of that page.

.SBTTL dtextd This statement defines the delimited string text to be the sub-title to be printed at the top of every page of the listing. It follows the same rules as the .TITLE pseudo-op.

.REMARK dtextd This statement inserts a remark into the listing. The delimited text can be any number of lines long, being terminated only by the matching delimiter.

.PRNTX dtextd This statement, when encountered, causes the delimited text string to be typed on the console. This statement is frequently used to print out conditional information, and to report the progress through pass 1 on very long assemblies.

Punch Control

The TDL Z80 Assembler normally produces an object tape in the TDL Standard Relocatable Format (see Appendix E). However, the assembler can produce an object tape compatible with the "INTEL Standard" hex tape. To control which format is being produced, the two pseudo-ops .PREL and .PABS are used. The .PABS pseudo-op causes the assembler to produce an INTEL compatible tape for all following generated code. The .PREL causes the assembler to return to producing TDL Standard Object Tape.

Every program starts with an implicit .PREL pseudo-op.

In addition, the assembler can punch the output tape in both binary and ASCII. To control which type of output is being produced, the two pseudo-ops .PBIN and .PHEX are used. The .PBIN pseudo-op causes the assembler to produce a binary tape in the current format. The .PHEX pseudo-op causes the output of an ASCII tape. Every program starts with an implicit .PHEX pseudo-op.

To control the generation of linkable object modules, two pseudo-ops are provided. The .LINK pseudo-op indicates that linkage information is to be included in the object file produced. The .XLINK pseudo-op inhibits this information from being output. Every program starts with an implicit .XLINK pseudo-op.

The TDL Z80 Assembler provides one additional facility to assist the TDL Z80 Debugging System. At the programmers option, the assembler will punch all of the global (non-local) symbols in the program module onto the end of the object tape. For each symbol, the assembler also punches its relocation base and its value relative to that base. Two pseudo-ops are provided to control this symbol table punching. The .PSYM pseudo-op enables the punching, and the .XPSYM pseudo-op disables it. The default is to not punch the symbol table (.XPSYM).

Conditional Assembly

Parts of a program may be assembled on a conditonal basis depending on the results of certain tests specified to the assembler through the use of the .IFx pseudo-op.

The general form of the pseudo-op is:

```
.IFx arg,[true text] ... {[false text]}
```

where the text within the first square brackets is assembled only if the specified test on the argument is TRUE, and the optional text within the second set of brackets is assembled if the condition is false. Any number of spaces or blank lines (or lines with only comments) may separate the true and false texts.

The square brackets around the true text may be omitted if there is no false text, and the entire true text is contained on the same line as the .IFx pseudo-op.

The first set of conditions which can be tested are the numeric value of the argument. These pseudo-ops are listed below:

```
.IFE  n,[...]      TRUE if n=0 or n=blank
.IFN  n,[...]      TRUE if n<0 or n>0
.IFG  n,[...]      TRUE if n>0
.IFGE n,[...]      TRUE if n>0 or n=0
.IFL  n,[...]      TRUE if n<0
.IFLE n,[...]      TRUE if n<0 or n=0
```

The following .IF pseudo-ops test for whether the assembler is processing pass 1 or not:

```
.IF1 ,[...]        TRUE if it is pass 1
.IF2 ,[...]        TRUE if it is not pass 1
```

The next set of conditionals tests for whether a symbol has been defined yet or not:

```
.IFDEF  symbol,[...]    TRUE if the symbol is defined
.IFNDEF symbol,[...]    TRUE if the symbol is undefined
```

The next set of .IF pseudo-ops tests to see whether its argument is blank or not. These pseudo-ops require that the argument be enclosed in square brackets ([]). The format is as follows:

```
.IFB  [...],[...]  TRUE if blank
.IFNB [...],[...]  TRUE if not blank
```

The quantity enclosed in the brackets is blank if it is empty, or consists only of spaces and tabs. Optionally, the argument being tested may be enclosed in paired delimiters

in the same manner as the .ASCIx pseudo-ops. If the first non-blank, non-tab, character after the pseudo-op is a left square bracket ([), the bracket method is used, otherwise, the delimiter method. For example:

```
.IFB /.../,[...]
```

The last pair of conditionals operate on character strings. They take two arguments which are interpreted as 7-bit ASCII character strings, and make a character by character comparison of the two strings to determine if the condition is met. Each of the strings may either be enclosed in square brackets or delimited by a character, as in the .IFB/.IFNB pseudo-ops above. The same method need not be used for both strings. The format of these conditionals is as follows:

```
.IFIDN [...] [...],[...]     TRUE if identical
.IFDIF [...] [...],[...]     TRUE if different
```

The maximum length of the strings to be compared is 255 characters. In making the comparison, all trailing blanks and tabs are ignored in the two arguments.

Synonyms

It sometimes becomes useful, for documentation or ease of programming, to define new names for already existing symbols. The TDL Z80 Assembler has four pseudo-ops which allow the definition of synonyms for already defined symbols. The format of these pseudo-ops is:

```
.xxSYN symbol1,symbol2
```

The four pseudo-ops are .SYN, .OPSYN, .SYSYN, and .MASYN. The only difference between the four is that the latter three limit the type of symbol for which the synonym is being defined.

The statement above defines the second operand as being synonymous with the first operand. In the case of the .SYN pseudo-op, the symbol tables are searched for the first operand in the order: programmer defined symbol, macro, operation. The .OPSYN pseudo-op limits the search to operations, the .SYSYN to programmer defined symbols, and the .MASYN to macros. The second operand is defined to be identical to the first operand at the time the synonym is defined. Later changes to the first operand will not affect the second.

The following are valid synonym definitions:

```
.OPSYN .BYTE,DB
.SYN .WORD,DW
```

```
.SYSYN ALPHA,BETA
.SYN A,R1
```

Object Machine Validation

Although the TDL Macro Assembler will run only on a Z80 processor, it can obviously be used to generate object code for any of the 8080 compatible micro-processors. To facilitate the use of the assembler for this purpose, two additional pseudo-ops are available: .I8080 and .Z80.

The .I8080 pseudo-op causes all subsequent uses of machine operations which are unique to the Z80 (and hence unavailable on the 8080) to be flagged with a Z warning message. Such uses will be properly assembled however.

The .Z80 pseudo-op (which is the default) disables the feature so that no further Z warnings will be given.

Chapter 4

Macros

A common characteristic of assembly language programs is that many coding sequences are repeated over and over with only a change in one or two of the operands. It is convenient, therefore, to provide a mechanism by which the repeated sequences can be generated by a single statement. The TDL Z80 Assembler provides the capability to do so by allowing the repeated sequences to be written, with dummy values for the changed operands, as a macro. A single statement, referring to the macro by name and providing values for the dummy operands, can then generate the repeated sequence.

Macro Definition

A macro is defined by use of the .DEFINE pseudo-op. This is followed by the symbolic name of the macro. The macro name must follow the rules for the construction of symbols. The name may be followed by a list of dummy arguments enclosed in square brackets. The dummy arguments are separated by commas, and may be any symbol which is convenient. Following the macro name and optional dummy arguments must be an equal sign (=). The following are examples of the heading part of a macro definition:

```
.DEFINE MACRO =
.DEFINE MOVE[A,B] =
.DEFINE BIGMAC[ARG1,ARG2,ARG3,%ARG5] =
```

Following the macro definition header comes the body of the macro. It need not start on the same line as the definition header. The body of the macro is delimited by a matched pair of left and right square brackets ([]). For example:

```
.DEFINE MOVE[A,B]=
[LDA A
STA B]
```

Macro Calls

A macro may be called by any statement. A macro call consists of the macro name followed (optionally) by a list of arguments. The arguments are separated by commas, and may optionally be enclosed in left and right square brackets ([]). If the brackets are used (the first non-blank/non-tab character after the macro name is a left square bracket),

then the arguments are terminated by a right square bracket. If there are n dummy arguments in the macro definition, then all arguments after the first n are ignored (although they do take space and time to process). If the brackets are omitted, the argument string ends when a carriage return or semicolon is encountered.

The arguments must be written in the order in which they are to be substituted for the dummy arguments. The first argument is substituted for each appearance of the first dummy argument, the second for the second, etc. The actual arguments are substituted as character strings for the dummy arguments, no evaluation of the arguments takes place until the macro is processed.

Referring to the definition of MOVE above, the occurrence of the statement:

```
MOVE ALPHA,BETA
```

will cause the substitution of ALPHA for A and BETA for B in the macro.

Statements which contain macro calls may be labelled and have comments like any other statement.

Macro arguments are terminated only by comma, carriage return, semicolon, or right square bracket (when started by left square bracket). These characters may not be used in the arguments unless the argument is enclosed in parentheses. Each time an argument is passed to a macro, one set of matched parentheses is removed, but all of the characters within the parentheses are substituted for the dummy argument in the macro. Note that spaces and tabs do not terminate arguments, but are considered to be part of them.

Macros do not need to have arguments. The macro name (and arguments if any) may appear anywhere in a statement where a symbol would normally appear, and the text of the macro exactly replaces the macro name and its arguments in that statement.

Comments

Comments may be included within a macro definition. Storing the comments with the macro (so that they will appear when the macro is expanded) takes space however. If the comment within the macro definition is preceded by two semicolons (instead of the normal one), the comment will be ignored during the definition of the macro, and will not be stored as part of the definition. This will eliminate the appearance of the comment every time the macro expansion is listed, however.

Created Symbols

When a macro is called, it is often useful to generate symbols without explicitly stating them in the call. A good example of this is labels within the macro body. It is usually not necessary to refer to these label externally to the macro expansion, therefore there is no reason why the programmer should be concerned as to what those labels are. The same with temporary data areas. To avoid conflicts, however, it is necessary that a different symbol be used each time the macro is called (even with local symbols, the macro could be called more than once between two global symbols). Created symbols are used for this purpose.

Each time a macro that requires a created symbol is called, a symbol is generated and inserted into the macro. These symbols are of the form ..nnnn (two periods followed by four digits). It should be noted that this makes these symbols local symbols (start with two periods). The programmer is advised not to use symbols of this form. The four digits start at 0000 and are incremented by one each time a symbol is created.

A created symbol is specified in the macro definition by preceding a dummy argument by a percent sign (%). When the macro is called, all dummy arguments of the form %symbol are replaced by created symbols (each with a different one). If, however, the position of the dummy argument in the argument list corresponds to an actual argument provided in the call, then the actual argument is used in place of the created one.

An actual argument can in fact be empty (signified by two consecutive commas in the argument list). An argument of this kind (a "null" argument) is considered to be defined as having a value of the empty string (no characters), and will prevent the generation of a created symbol for its corresponding dummy argument.

For example:

```
.DEFINE PRINT[A,%B]=
[CALL LINPRT
JMP %B
.ASCIS \A\
%B:]
```

This macro prints a message on the printer. The first argument to the macro is the text string to be printed. LINPRT is a line printer routine. Labelling the location following the text is necessary because of the indeterminate length of the message. The use of a created symbol here is useful since there would normally be no reason to reference the label. Calling the macro by:

```
PRINT This is the message
```

would result in printing "This is the message" when the assembled macro was executed. If it had been called:

```
PRINT This is the message,MAIN
```

the message would have been printed, but control would be transferred to the label MAIN, which substituted for %B instead of a created symbol.

Concatenation

The apostrophe or single quote (') is defined within a macro definition as the concatenation operator. This allows a macro argument to be only part of a symbol or expression, with the character string which is substituted for the dummy argument being joined with other character strings that are part of the macro definition to form a complete symbol or expression. This joining is called concatenation. Concatenation is performed by the assembler when an apostrophe is used between the strings to be joined (one or both of which must be a dummy macro argument). For example:

```
.DEFINE BR[A,B]=
[JR'A B]
```

defines a conditional branch statement. When called, the argument A is appended to the JR to form a single symbol. If the call were:

```
BR Z,LOOP
```

then the generated code would be:

```
JRZ LOOP
```

Default Arguments

Normally, missing arguments in a macro are replaced by nulls. For example, in the macro:

```
.DEFINE BYTES[A1,A2,A3,A4,A5,A6]=
[.BYTE A1,A2,A3,A4,A5,A6]
```

a call of BYTES[1,2] would generate an error because of the missing arguments to the pseudo-op .BYTE.

To remedy this, the assembler provides the programmer with the means to supply default arguments to be used when no argument is provided in the macro call. Default arguments are defined as part of the macro definition by enclosing them in parentheses and inserting them immediately

after the dummy argument to which they refer. To solve the above problem, the definition would be written as:

```
.DEFINE BYTES[A1(0),A2(0),A3(0),A4(0),A5(0),A6(0)]=
[.BYTE A1,A2,A3,A4,A5,A5]
```

which would always generate six bytes of data, regardless of how many arguments were provided in the call.

ASCII Interpretation of Numeric Arguments

If the reverse slash (\) preceeds the first character of an argument in a macro call, the value of the expression following the reverse slash is converted to an ASCII string. This string is then used as the argument to the call. The value is considered to be a 16-bit positive value, and the conversion is done in the current radix. Leading zeros are suppressed unless the value is zero.

For example:

```
A = 5
B = 6
MACRO \A+B, \A*B
```

is the same as:

```
MACRO 11, 30
```

if the current radix is 10.

Macro Expansion Termination

Under normal conditions, a macro expansion terminates at the end of the macro definition. It is sometimes desirable to terminate the macro expansion prior to the end of the definition. This is usually done as part of some conditional assembly within the macro. A special pseudo-op is provided for this purpose:

```
.EXIT
```

When processed by the assembler, the .EXIT pseudo-op immediately terminates the macro expansion, just as if the end of the macro had been encountered. Only the current expansion is terminated if multiple macro expansions are being nested.

User Defined Macro Errors

It is sometimes desirable to have a macro cause an assembly error. This might be done when invalid parameters are passed to the macro, or if parameters are missing. A

special pseudo-op is provided to allow this:

.ERROR dtextd

This pseudo-op will cause an asterisk (*) to be listed as the error code, the error count to be incremented by one, and the line to be listed as an error. The delimited text is treated exactly as in a .REMARK pseudo-op, and can be used to provide information about the nature of the error.

Nesting

Macros may be nested. This means that macros may be both called and defined within other macros. A macro that is defined within another macro may not be called until the defining macro has been called. At that time, the new macro is available to be called by any statement.

The only limit to how many levels deep macro calls and definitions may be nested is the amount of memory available.

Appendix A

Summary of Machine Operation Mnemonics

The following section presents a summary of the Z80 machine operations and their assembler mnemonics. The appendix is arranged by type of instruction for ease of reference. For further information on the machine operations, refer to the "ZILOG Z80-CPU Technical Manual".

To make the information presented more readily usable, a shorthand notation is used for describing the assembler format of the instruction and its actual operation. All capital letters and special charcters in the mnemonic description are required. The lower case letters indicate a class of values which can be inserted in the instruction at that point. A single lower case letter indicates an 8-bit quanity or register, while a double lower case letter indicates a 16-bit quantity or register. A symbol enclosed in parentheses in the machine operation section indicates that the value whose address is specified is used. The following is a summary of the notation used; exceptions will be noted where appropriate in the following sections.

r	one of the 8-bit registers A, B, C, D, E, H, L
n	any 8-bit absolute value
ii	an index register reference, either X or Y
d	an 8-bit index displacement where -128 < d < 127
zz	B for the BC register pair, D for the DE pair
nn	any 16-bit value, absolute or relocatable
rr	B for the BC register pair, D for the DE pair, H for the HL pair, SP for the stack pointer
qq	B for the BC register pair, D for the DE pair, H for the HL pair, PSW for the A/Flag pair
s	any of r (defined above), M, or d(ii)
IFF	interupt flip-flop
CY	carry flip-flop
ZF	zero flag
tt	B for the BC register pair, D for the DE pair, SP for the stack pointer, X for index register IX
uu	B for the BC register pair, D for the DE pair, SP for the stack pointer, Y for index register IY
b	a bit position in an 8-bit byte, where the bits are numbered from right to left 0 to 7
PC	program counter
v[n]	bit n of the 8-bit value or register v
v[n-m]	bits n through m of the 8-bit value or register v
vv\H	the most significant byte of the 16-bit value or register vv
vv\L	the least significant byte of the 16-bit value or register vv

Iv an input operation on port v
Ov an output operation on port v
w<-v the value of w is replaced by the value of v
w<->v the value of w is exchanged with the value of v

8-Bit Load Group

Mnemonic		Operation	# of Bytes
MOV	r,r'	r <- r'	1
MOV	r,M	r <- (HL)	1
MOV	r,d(ii)	r <- (ii+d)	3
MOV	M,r	(HL) <- r	1
MOV	d(ii),r	(ii+d) <- r	3
MVI	r,n	r <- n	2
MVI	M,n	(HL) <- n	2
MVI	d(ii),n	(ii+d) <- n	4
LDA	nn	A <- (nn)	3
STA	nn	(nn) <- A	3
LDAX	zz	A <- (zz)	1
STAX	zz	(zz) <- A	1
LDAI		A <- I	2
LDAR		A <- R	2
STAI		I <- A	2
STAR		R <- A	2

16-Bit Load Group

Mnemonic		Operation	# of Bytes
LXI	rr,nn	rr <- nn	3
LXI	ii,nn	ii <- nn	4
LBCD	nn	B <- (nn+1) C <- (nn)	4
LDED	nn	D <- (nn+1) E <- (nn)	4
LHLD	nn	H <- (nn+1) L <- (nn)	3
LIXD	nn	IX\H <- (nn+1) IX\L <- (nn)	4
LIYD	nn	IY\H <- (nn+1) IY\L <- (nn)	4
LSPD	nn	SP\H <- (nn+1) SP\L <- (nn)	4
SBCD	nn	(nn+1) <- B (nn) <- C	4
SDED	nn	(nn+1) <- D (nn) <- E	4
SHLD	nn	(nn+1) <- H (nn) <- L	3
SIXD	nn	(nn+1) <- IX\H (nn) <- IX\L	4
SIYD	nn	(nn+1) <- IY\H (nn) <- IY\L	4
SSPD	nn	(nn+1) <- SP\H (nn) <- SP\L	4
SPHL		SP <- HL	1
SPIX		SP <- IX	2
SPIY		SP <- IY	2
PUSH	qq	(SP-1) <- qq\H (SP-2) <- qq\L SP <- SP - 2	1
PUSH	ii	(SP-1) <- ii\H (SP-2) <- ii\L SP <- SP - 2	2
POP	qq	qq\H <- (SP+1) qq\L <- (SP) SP <- SP + 2	1
POP	ii	ii\H <- (SP+1) ii\L <- (SP) SP <- SP + 2	2

Exchange and Block Transfer and Search Group

Mnemonic	Operation	# of bytes
XCHG	HL <-> DE	1
EXAF	PSW <-> PSW'	1
EXX	BCDEHL <-> BCDEHL'	1
XTHL	H <-> (SP+1) L <-> (SP)	1
XTIX	IX\H <-> (SP+1) IX\L <-> (SP)	2
XTIY	IY\H <-> (SP+1) IY\L <-> (SP)	2
LDI	(DE) <- (HL) DE <- DE + 1 HL <- HL + 1 BC <- BC - 1	2
LDIR	repeat LDI until BC=0	2
LDD	(DE) <- (HL) DE <- DE - 1 HL <- HL - 1 BC <- BC - 1	2
LDDR	repeat LDD until BC=0	2
CCI	A - (HL) HL <- HL + 1 BC <- BC - 1	2
CCIR	repeat CCI until A=(HL) or BC=0	2
CCD	A - (HL) HL <- HL - 1 BC <- BC - 1	2
CCDR	repeat CCD until A=(HL) or BC=0	2

8-Bit Arithmetic and Logical Group

Mnemonic		Operation	# of Bytes
ADD	r	A <- A + r	1
ADD	M	A <- A + (HL)	1
ADD	d(ii)	A <- A + (ii+d)	3
ADI	n	A <- A + n	2
ADC	s	A <- A + s + CY	
ACI	n		
SUB	s	A <- A - s	
SUI	n		
SBB	s	A <- A - s - CY	
SBI	n		
ANA	s	A <- A & s	
ANI	n		
ORA	s	A <- A ! s	
ORI	n		
XRA	s	A <- A ^ s	
XRI	n		
CMP	s	A - s	
CPI	n		
INR	s	s <- s + 1	
DCR	s	s <- s - 1	

General Purpose Arithmetic and Control Group

Mnemonic	Operation	# of Bytes
DAA	convert A to packed BCD after an add or subtract of packed BCD operands	1
CMA	A <- #A	1
NEG	A <- -A	2
CMC	CY <- #CY	1
STC	CY <- 1	1
NOP	no operation	1
HLT	halt	1
DI	IFF <- 0	1
EI	IFF <- 1	1
IM0	interrupt mode 0	2
IM1	interrupt mode 1	2
IM2	interrupt mode 2	2

16-Bit Arithmetic Group

Mnemonic		Operation	# of Bytes
DAD	rr	HL <- HL + rr	1
DADC	rr	HL <- HL + rr + CY	2
DSBC	rr	HL <- HL - rr - CY	2
DADX	tt	IX <- IX + tt	2
DADY	uu	IY <- IY + uu	2
INX	rr	rr <- rr + 1	1
INX	ii	ii <- ii + 1	2
DCX	rr	rr <- rr - 1	1
DCX	ii	ii <- ii - 1	2

Rotate and Shift Group

Mnemonic		Operation	# of Bytes
RLC		A[n+1] <- A[n] A[0] <- A[7] CY <- A[7]	1
RAL		A[n+1] <- A[n] A[0] <- CY CY <- A[7]	1
RRC		A[n] <- A[n+1] A[7] <- A[0] CY <- A[0]	1
RAR		A[n] <- A[n+1] A[7] <- CY CY <- A[0]	1
RLCR	s	s[n+1] <- s[n] s[0] <- s[7] CY <- s[7]	2 (or 4)
RALR	s	s[n+1] <- s[n] s[0] <- CY CY <- s[7]	
RRCR	s	s[n] <- s[n+1] s[7] <- s[0] CY <- s[0]	
RARR	s	s[n] <- s[n+1] s[7] <- CY CY <- s[0]	
SLAR	s	s[n+1] <- s[n] s[0] <- 0 CY <- s[7]	
SRAR	s	s[n] <- s[n+1] s[7] <- s[7] CY <- s[0]	
SRLR	s	s[n] <- s[n+1] s[7] <- 0 CY <- s[0]	
RLD		A[0-3] <- (HL)[4-7] (HL)[4-7] <- (HL)[0-3] (HL)[0-3] <- A[0-3]	2
RRD		(HL)[0-3] <- (HL)[4-7] (HL)[4-7] <- A[0-3] A[0-3] <- (HL)[0-3]	2

Bit Set, Reset, and Test Group

Mnemonic		Operation	# of Bytes
BIT	b,r	ZF <- #r[b]	2
BIT	b,M	ZF <- #(HL)[b]	2
BIT	b,d(ii)	ZF <- #(ii+d)[b]	4
SET	b,s	s[b] <- 1	
RES	b,s	s[b] <- 0	

Jump Group

Mnemonic		Operation	# of Bytes
JMP	nn	PC <- nn	3
JZ	nn	if zero, then JMP else continue	3
JNZ	nn	if not zero	3
JC	nn	if carry	3
JNC	nn	if not carry	3
JPO	nn	if parity odd	3
JPE	nn	if parity even	3
JP	nn	if sign positive	3
JM	nn	if sign negative	3
JO	nn	if overflow	3
JNO	nn	if not overflow	3
JMPR	nn	PC <- nn where -126 < nn-PC < 129	2
JRZ	nn	if zero, then JMPR else continue	2
JRNZ	nn	if not zero	2
JRC	nn	if carry	2
JRNC	nn	if not carry	2
DJNZ	nn	B <- B - 1 if B=0 then continue else JMPR	2
PCHL		PC <- HL	1
PCIX		PC <- IX	2
PCIY		PC <- IY	2

Call and Return Group

Mnemonic		Operation	# of Bytes
CALL	nn	(SP-1) <- PC\H (SP-2) <- PC\L SP <- SP - 2 PC <- nn	3
CZ	nn	if zero, then CALL else continue	3
CNZ	nn	if not zero	3
CC	nn	if carry	3
CNC	nn	if not carry	3
CPO	nn	if parity odd	3
CPE	nn	if parity even	3
CP	nn	if sign positive	3
CM	nn	if sign negative	3
CO	nn	if overflow	3
CNO	nn	if not overflow	3
RET		PC\H <- (SP+1) PC\L <- (SP) SP <- SP + 2	1
RZ		if zero, then RET else continue	1
RNZ		if not zero	1
RC		if carry	1
RNC		if not carry	1
RPO		if parity odd	1
RPE		if parity even	1
RP		if sign positive	1
RM		if sign negative	1
RO		if overflow	1
RNO		if no overflow	1
RETI		return from interrupt	2
RETN		return from non-maskable interrupt	2
RST	n	(SP-1) <- PC\H (SP-2) <- PC\L PC <- 8 * n where 0 <= n < 8	1

Input and Output Group

Mnemonic		Operation	# of Bytes
IN	n	A <- In	2
INP	r	r <- I(C)	2
INI		(HL) <- I(C) B <- B - 1 HL <- HL + 1	2
INIR		repeat INI until B=0	2
IND		(HL) <- I(C) B <- B - 1 HL <- HL - 1	2
INDR		repeat IND until B=0	2
OUT	n	On <- A	2
OUTP	r	O(C) <- r	2
OUTI		O(C) <- (HL) B <- B - 1 HL <- HL + 1	2
OUTIR		repeat OUTI until B=0	2
OUTD		O(C) <- (HL) B <- B - 1 HL <- HL - 1	2
OUTDR		repeat OUTD until B=0	2

Appendix B

Summary of Pseudo-Operation Mnemonics

.ASCII dtextd | [n] ...

The .ASCII pseudo-op enters 7-bit ASCII characters into the program. The text is either entered between two delimiters, or as a numeric value enclosed in square brackets ([]), and the two forms may be intermixed and repeated as desired.

.ASCIS dtextd | [n] ...

The .ASCIS pseudo-op enters 7-bit ASCII characters into the program, and flags the last character by setting its high-order bit on. The format of the text is the same as for the .ASCII pseudo-op.

.ASCIZ dtextd | [n] ...

The .ASCIZ pseudo-op enters 7-bit ASCII characters into the program, and flags the end of the characters by inserting an additional null byte. The format of the text is the same as for the .ASCII pseudo-op.

.BLKB nn

The .BLKB pseudo-op reserves a block of contiguous storage nn bytes long.

.BLKW nn

The .BLKW pseudo-op reserves a block of contiguous storage nn words long (nn x 2 bytes).

.BYTE n {, n ...}

The .BYTE pseudo-op enters single byte values into the program. Multiple values may be entered by separating them with a comma.

.DEFINE symbol[arg1,arg2,...]=[text]

The .DEFINE pseudo-op defines a macro with the name symbol. arg1 through argn are optional dummy arguments. The body of the macro is represented by text.

.END nn

The .END pseudo-op signals the end of the assembly. When encountered during PASS 1, it simply returns to the initialization section. During a listing pass, it initiates the listing of the symbol table (if not previously suppressed by the .XSYM pseudo-op). During a punch pass, it generates an EOF record on the hex tape containing the value nn as the starting address of the object program.

.ENTRY symbol1 {, symbol2 ...}

The .ENTRY pseudo-op identifies the internally defined symbols which are subroutine library entry points to this program. Multiple symbols may be identified by separating them with commas.

.ERROR dtextd

The .ERROR pseudo-op causes an "*" error to occur, forcing the listing of the current line, and an error notification. The delimited text is treated as a .REMARK.

.EXIT

The .EXIT pseudo-op causes an immediate exit from the current macro expansion to occur.

.EXTERN symbol1 {, symbol2 ...}

The .EXTERN pseudo-op defines those symbols which are referenced in this program but are defined in another, separately assembled, program. Multiple symbols can be defined by separating them with commas.

.I8080

The .I8080 pseudo-op enables the Z warning message. This warning will be given whenever a machine operation unique to the Z80 is encountered.

.IDENT symbol

The .IDENT pseudo-op gives the module a name for later use by the linkage editor.

.INTERN symbol1 {, symbol2 ...}

The .INTERN pseudo-op identifies those symbols which are defined in this program and which will be referenced as external symbols by some separately assembled program. Multiple symbols may be identified by separating them with commas.

.LADDR

The .LADDR pseudo-op changes the listing mode from displaying 16-bit quantities to displaying the Z80 image with the least significant byte first.

.LALL

The .LALL pseudo-op causes the assembler to list every text character processed, including those suppressed in the normal listing.

.LCTL

The .LCTL pseudo-op causes the assembler to list all listing control statements.

.LINK

The .LINK pseudo-op causes the assembler to output linkage information to the object file.

.LIST

The .LIST pseudo-op resumes a listing which has been stopped by the .XLIST pseudo-op.

.LIMAGE

The .LIMAGE pseudo-op changes the listing mode to display every byte of object code generated rather than the normal mode of a maximum of five bytes per statement.

.LOC nn

The .LOC pseudo-op changes the value of the assembler's program counter to nn. If nn is relocatable, then all labels will be assigned relocatable values. If it is absolute, then absolute values will be assigned.

.LSYM

The .LSYM pseudo-op reenables the listing of the symbol table during the .END pseudo-op processing after it has been disabled by the .XSYM pseudo-op. The .LSYM pseudo-op must occur prior to the .END pseudo-op to be effective.

.MASYN symbol1,symbol2

The .MASYN pseudo-op allows the definition of a new macro to be the same as a previously defined one. Symbol2 is defined to be a macro identical to the one defined as symbol1.

.OPSYN symbol1,symbol2

The .OPSYN pseudo-op allows the definition of a new op code mnemonic as a synonym of an already existing one. The symbol1 must be a defined machine or pseudo op code (or one previously defined using .OPSYN), symbol2 will be defined to be the same operation.

.PABS

The .PABS pseudo-op signals that the hex object tape produced from this point on in the assembly is to be in absolute (INTEL compatable) format.

.PAGE

The .PAGE pseudo-op causes a skip to the top of the next page during a listing pass.

.PBIN

The .PBIN pseudo-op specifies that the object tape is to be produced in binary.

.PHEX

The .PHEX pseudo-op specifies that the object tape is to be produced in ASCII.

.PREL

The .PREL pseudo-op signals that the hex object tape produced from this point on in the assembly is to be in relocatable (TDL standard) format.

.PRNTX dtextd

The .PRNTX pseudo-op will cause its text string to be printed on the console whenever it is encountered in the assembly process.

.PSYM

The .PSYM pseudo-op signals that the entire symbol table from the assembly is to be punched at the end of the object tape. The .PSYM pseudo-op must appear prior to the .END pseudo-op to be effective.

.RADIX n

The .RADIX pseudo-op changes the default base in which a numeric constant is interpreted during the assembly to n. The valid values for n are 2, 8, 10, or 16. The value is always interpreted as a decimal number.

.RAD40 symbol

The .RAD40 pseudo-op generates a unique 4 byte value in radix-40 notation for the symbol given. The symbol must conform to the rules for any symbol in the assembly. This pseudo-op is used mostly for developing system software utilizing symbol tables.

.RELOC

The .RELOC pseudo-op restores the value of the assembler's program counter to whatever it was before the immediately preceding .LOC pseudo-op.

.REMARK dtextd

The .REMARK pseudo-op allows the entry of multiple line comments into the source program. All of the text between the delimiters is listed but is ignored. The text may contain carriage return/line feeds.

.RLIST

The .RLIST pseudo-op restores the listing control flags from the top element of the .SLIST push-down stack.

.SALL

The .SALL pseudo-op suppresses all macro expansions on the assembly listing (normally all lines generating code are listed).

.SBTTL dtextd

The .SBTTL pseudo-op sets the sub-title for the assembly listing to the specified text string (which must be less than 72 characters in length). If the .SBTTL pseudo-op is the first operation after a .PAGE, the sub-title will appear on the new page.

.SLIST

The .SLIST pseudo-op saves the current listing control flags on the top of a four element push-down stack.

.SYN symbol1,symbol2

The .SYN pseudo-op makes any two symbols synonymous. The symbol tables are searched for symbol1 in the normal operand field order (label/symbol, macro, opcode), and symbol2 is defined to have the same value as symbol1.

.SYSYN symbol1,symbol2

The .SYSYN pseudo-op makes one symbol the synonym of an already defined symbol/label. The value of a symbol/label symbol1 is obtained, and symbol2 is defined to be the same type and value.

.TITLE dtextd

The .TITLE pseudo-op sets the title for the assembly listing to the specified text string (which must be less than 72 characters in length). The title is put at the top of every page during a listing. If the .TITLE pseudo-op is the first operation after a .PAGE pseudo-op, the title will be listed on the new page.

.WORD nn {, nn ...}

The .WORD pseudo-op enters 2-byte values into the program in proper Z80 format (least significant byte first). Multiple values may by entered by separating them with a comma.

.XADDR

The .XADDR pseudo-op is used after a .LADDR pseudo-op to return to the standard format of listing 16-bit values.

.XALL

The .XALL pseudo-op is used after a .LALL or .SALL pseudo-op to return to the standard listing mode.

.XCTL

The .XCTL pseuod-op is used after a .LCTL pseudo-op to return the standard mode of suppressing the listing of listing control statements.

.XIMAGE

The .XIMAGE pseudo-op is used after a .LIMAGE pseudo-op to return to the standard listing mode of only five object bytes per statement.

.XLINK

The .XLINK pseudo-op is used after a .LINK pseudo-op to suppress the inclusion of linkage information in the object file.

.XLIST

The .XLIST pseudo-op suppresses the listing of all following statements (until a .LIST pseudo-op is encountered).

.XPSYM

The .XPSYM pseudo-op disables the punching of the symbol table at the end of the object tape after it has been enabled by the .PSYM pseudo-op. The .XPSYM pseudo-op must occur prior to the .END pseudo-op to be effective.

.XSYM

The .XSYM pseudo-op disables the listing of the symbol table by the .END pseudo-op (unless reenabled by the .LSYM pseudo-op). The .XSYM pseudo-op must appear before the .END pseudo-op to be effective.

.Z80

The .Z80 pseudo-op is used to disable the effect of a previous .I8080 pseudo-op. This inhibits the Z warning message on machine operations unique to the Z80.

.IFx arg,[true text] ... {[false text]}

The .IFx pseudo-op will assemble the true text specified only if the particular condition being tested for is true, The optional false text is assembled if the condition is false. The .IFx pseudo-ops and their conditions are as follows:

.IF1: assembling pass 1
.IF2: not assembling pass 1
.IFB: blank
.IFDEF: defined
.IFDIF: different
.IFE: zero or blank
.IFG: positive
.IFGE: zero or positive
.IFIDN: identical
.IFL: negative
.IFLE: zero or negative
.IFN: not zero
.IFNB: not blank
.IFNDEF: not defined

Appendix C

Operation of the Assembler with a TDL Monitor

The TDL Z80 Relocating Assembler is designed to operate with a TDL System Monitor. It relies upon the Monitor for all I/O and memory management functions. (For further information on the TDL Monitors, consult the appropriate monitor reference manual.) When operating, the assembler will use all available memory for its various tables (all memory between the end of the assembler and the highest available memory location). No memory location below the assembler is changed by its operation.

The first step in using the assembler is to load it into the desired memory location using the monitor "R" command. After the load has been completed, if the monitor is not located at the standard memory address (F000 hex), it will be necessary to change the assembler's monitor transfer vector to point to the monitor. This transfer vector consists of nine (9) JMP instructions located beginning at relative address six (6 hex) in the program. The addresses of these instructions should be modified to point to the correct locations.

After the assembler is loaded and ready to operate, the appropriate monitor commands should be used to designate the reader, punch, and list devices as desired. The console device is also used during the assembly. After readying the source program in the reader, a "G" command should be used to start the assembler.

It is important to note that the assembler requires a "controlled" reader device (a device which provides characters on demand, at whatever rate the program wants them). In the same manner in which the assembler "waits" for the next character from the reader, the reader must be capable of "waiting" for the next demand from the assembler. (For further information on converting a non-controlled reader to a controlled one, see one of the TDL System Monitor reference manuals.)

When first started (and whenever an assembly pass is completed), the assembler asks "PASS=" on the console. Valid responses to this are only the numbers from 0 to 8. A response of 0 will return to the monitor, but in a manner which will allow resumption of the assembly by reentering the "G" command. The values 1 through 4 signify which assembler pass is desired, as follows:

1 signifies the first assembly pass. The source is read, and all necessary tables are built.

2 signifies the listing only pass. The source is re-read, and a listing of the assembled program is produced on the list device.

3 signifies the punch only pass. The source is re-read, and an object tape of the assembled program is produced on the punch device.

4 signifies the combination of passes 2 and 3.

The values of 5 through 8 provide the same options as 1 through 4, but do not reinitialize the assembler in any way before proceeding. This allows the assembly of a program residing on more than one source tape. Each of the pieces must, however, be terminated by its own .END pseudo-op.

During the first assembly pass (pass 1), it is possible that some error messages will be output on the list device. These errors will be those uniquely determined during the pass.

During the punch only pass (pass 3), no error messages will be listed, but an errors indication will be given on the console at the end of the assembly.

While an assembly is taking place, a number of console control options are available. A control-C will always trap back to the monitor after the completion of the current statement. The assembly may be resumed (if no registers have been changed) by using the monitor "G" command. A control-C will, however, result in monitor output on the console device, which could spoil a listing if the console is the list device. To avoid this, the use of a control-S will temporarily halt the assembly (e.g. to put more paper in the teletype), but will not return to the monitor or cause any spurious output on the console device. A control-Q will resume the assembly. If a control-C is entered after the control-S, a trap to the monitor will occur as above. In addition, a control-T may be used to stop the assembly at the top of the next output page of the listing. When the control-T is entered on the keyboard, nothing will happen until the top-of-page is reached, at which time the assembler will act as if a control-S had been entered (see above). All of the above features will, however, be disabled if the reader device is specified as the Teletype.

When starting a listing pass, the paper in the list device should be positioned at the top line of a page. The assembler will count lines and put a page number and heading at the top of every page. The page width is determined by the assigned list device. If the list device is the teletype (AL=T), then the page is assumed to be 72 characters wide. If not, then it is assumed to be 80 characters wide. In either case, it is assumed to be 66 lines long, and a two line margin is left at the top and the

bottom of the page.

Appendix D

Error Codes

Errors in the source program encountered during the assembly process are indicated on the listing by a single letter code at the left of the statement in error. Although the assembler may detect more than two errors per statement, only the first two codes are given. As an added aid to locating the error in the statement, a question mark is printed to the right of the character which triggered the error. All errors generate a question mark, even if they are not one of the first two per statement.

The following is a list of the error codes and their meanings:

A Argument error. This is a broad class of errors which may be caused by many different things.

B Bad macro error. Either an error in a macro definition or a call on a bad macro.

D Duplicate symbol reference error. The symbol flagged is multiply-defined. The first value given to the symbol is used in the assembly.

E External symbol error. An external symbol is improperly used in the statement.

I Internal symbol error. An internal symbol is improperly used in the statement.

L Label error. An invalid character has been found in the label field of the statement.

M Multiply-defined symbol error. A symbol is defined more than once. This error is given mostly during Pass 1. During the other passes, it usually will appear as a phase error (P).

O Operation error. The symbol in the operation field is not a valid machine operation code, macro name, or symbol.

P Phase error. A label is assigned a value during Pass 2 (or 3 or 4) which is different than that assigned during Pass 1.

Q Questionable error. This is a broad class of warnings which the assembler gives when it finds ambiguous

statements. Q errors may or may not generate correct code. The assembler will attempt to do what the programmer intended.

R Relocation error. A relocatable symbol or expression is incorrectly used (eg. in a .BLKB pseudo-op).

T Table overflow. One of the Assembler's internal tables has overflowed. The Assembler will attempt to continue, but no new labels or macros will be defined.

U Undefined label/symbol error. A symbolic reference which was never defined is used in the statement.

X Index error. Another character appears in a statement at a point where only an index register reference is allowed (X or Y).

Z Z80 error. A Z80 machine operation has been encountered while in 8080 mode (.I8080). This is only a warning and the opcode will be properly assembled.

* User defined macro error. A .ERROR pseudo-op was encountered.

Appendix E

Object Tape Formats

The TDL Assembler produces two different object tape formats depending on the use of the .PABS and the .PREL pseudo-ops. It also punches the two formats two different ways, binary (.PBIN) and ASCII (.PHEX). Each of the two formats will be described separately, and where differences between binary and ASCII exist, they will be noted. In addition, the .XLINK option allows the suppression of some of the information in the relocatable format to allow the direct production of a relocatable core image module instead of a relocatable object module.

TDL Object Module Format Definition

The use of the .PREL pseudo-op (which is default if neither is specified) causes the generation of the TDL Object Module Format. This format allows for simple relocation of complete programs by the TDL System Monitors, and for complex relocation and linking of modules by the TDL Linkage Editor.

The default object module format is an extension of the INTEL "hex file" format, but is not compatible with that format. The module consists of a sequential file of ASCII characters representing the binary data, symbol, and control information required to construct a final program from the module. All binary bytes within this structure are represented as two ASCII characters corresponding to the hexadecimal value of the byte (e.g. 11001001 -> C9). All ASCII values are represented by the corresponding ASCII character (e.g. A -> A). In the binary punch mode, the format is basically the same, but all binary bytes are represented by themselves, not as two ASCII characters.

Each of the different records within the module is indicated by the use of a prompt character as the first character of the record (in the INTEL format, this is the ":"). The valid prompt characters are:

```
! -> module identification record
@ -> entry point record
# -> internal symbol record
\ -> external symbol/relocation base record
& -> symbol table record
; -> data/program/end-of-file record
```

(Note that only the records prompted by a ; are output if the .XLINK mode is in effect.)

Every record in the module is terminated by a one byte binary checksum of all of the preceeding bytes in the record except for the prompt character. The checksum is the two's complement of the sum of the preceeding bytes. Any output format (two character binary, one character ASCII or one byte binary) still counts as only one byte in the checksum (i.e. before conversion for output).

In addition, each record in the ASCII punch mode is preceeded by a carriage return/line feed sequence to facilitate listing the module on an external device. It is not present in the binary punch mode.

The following descriptions are specified assuming ASCII punch mode. With the above noted exception of the carriage return/line feed preceeding each record, the binary format is identical, with each binary byte being left unexpanded. ASCII characters are left as they are in either mode.

Module Identification Record (!)

Byte	1-2	CR/LF
	3	Exclamation point (!) prompt.
	4-9	ASCII module name.
	10-11	Checksum.

Entry Point Record (@)

Byte	1-2	CR/LF
	3	At-sign (@) prompt.
	4-5	Number of entry points in this record.
	6-??	ASCII names of entry points, 6 bytes per name. The names are left justified and blank filled.
	??	Checksum

Internal Symbol Record (#)

Byte	1-2	CR/LF
	3	Pound sign (#) prompt.
	4-5	Number of internal symbols in this record.
	6-11	ASCII name of internal symbol, left justified and blank filled.
	12-13	Relocation base for symbol. The value of this

symbol is relative to the relocation base specified.
14-17 Symbol value (16 bit).
.... The above three fields are repeated for each internal symbol in the record.
?? Checksum.

External Symbol/Relocation Base Record (\)

Byte 1-2 CR/LF
3 Back-slash (\) prompt.
4-5 Number of external/relocation symbols in this record.
6-11 ASCII name of the symbol, left justified and blank filled.
12-13 Relocation number assigned to this symbol in this module. This number is unique for each symbol. It starts with one and increases sequentially for each subsequent external/relocation base symbol.
14-17 Relocation segment size/external reference flag. If this value is zero, it represents a reference to a symbol defined externally to this module (usually a subroutine or global data item). If it is non-zero, then the value is the size of the relocation segment as defined in this object module. This segment can contain either code or data, and may be located anywhere in memory by the linkage editor, independent of any other segment.
.... The above three fields are repeated for each symbol contained in this record.
?? Checksum.

Symbol Table Record (&)

Byte 1-2 CR/LF
3 Ampersand (&) prompt.
4-?? The remainder of this record is identical to the internal symbol record. All symbols defined in this module are contained in these records.

Data/Program Record (;)

Byte 1-2 CR/LF
3 Semicolon (;) prompt

4-5	Number of binary data bytes in this record. The maximum is 32 binary bytes (64 bytes of ASCII representation). If this value is zero, this record is a end-of-file record, described below.
6-9	Load address of the data relative to the specified relocation base.
10-11	Relocation base for all relocation in this record. All relocatable values in this record are added to the current value of the specified relocation base before being put into memory. (If .XLINK is in effect, the only allowable relocation bases are 0 and 1.)
12-13	Relocation control byte. This byte controls the relocation of the next eight bytes in the record (if that many remain according to the count field). The bits are used from left to right. The bits have the following meanings: 0: a single absolute byte -> load unmodified. 10: a two byte relocatable value, least significant byte first -> add the 16 bit value to the current relocation base, and load the result least significant byte first. (If .XLINK is in effect, and the current relocation base is 0, then the 16 bit value is added to relocation base 1.) 110: a three byte reference to a different relocation base. The first byte is the relocation base number, and the two after that are the 16 bit value, least significant byte first -> add the specified relocation base to the 16 bit value, and load the result least significant byte first. (In .XLINK mode, this control pattern is not generated.) Note that a two or three byte combination is never broken across a record boundary.
14-29	Data bytes controlled as above.
30-??	The above control/data byte combinations are repeated as specified by the count.
??	Checksum.

End-of-File Record (;)

Byte	1-2	CR/LF
	3	Semicolon (;) prompt.
	4-5	Zero to indicate end-of-file record.
	6-9	Starting address for module relative to the

specified relocation base. This address is optionally generated by the language processor, and may be zero.

10-11 Relocation base for starting address. (In .XLINK mode may be only 0 or 1.)

12-13 Checksum.

INTEL Object Format

The use of the .PABS pseudo-op causes an INTEL "hex" object module to be produced. This object tape can also be loaded by the TDL System Monitors, but provides no relocatability.

All of the above comments concerning byte formats and checksums apply to this format as well.

Byte 1-2 CR/LF

3 Colon (:) prompt.

4-5 Number of binary data bytes in this record. The maximum number is 32 binary bytes (64 bytes of ASCII representation). If this value is zero, this record is an end-of-file record, and the load address is the program starting address.

6-9 Load address of the data in this record.

10-11 Unused.

12-?? Data bytes.

?? Checksum.

Appendix F

Additional Capabilities under CP/M

Library File Generation

It is often desirable to maintain a related set of independent object modules as a single source and object file for later use with the library search facility of the TDL Linkage Editor. To facilitate this the .PRGEND pseudo-op can be used. The format is:

.PRGEND

This pseudo-op functions identically to the .END pseudo-op, except that, after completing the assembly of the current module, the assembler continues with another module following. Multiple modules assembled in this manner from a single source file produce a single object file which can be linked in library search mode, and a single listing. Each module assembly is completely independent however. The last module in the source file must be terminated by a .END pseudo-op, not a .PRGEND.

Library Source File Usage

It is often convenient to be able to utilize the same section of assembler source code in a number of different assemblies. The .INSERT pseudo-op allows this to be done easily. The format is:

.INSERT {d:}file{.ext}

where d is the optional CP/M disk specifier (defaulting to the source file disk), file is the desired file name, and ext is the optional file extension (defaulting to ASM).

This pseudo-op causes the specified file to be copied into the assembly in its entirety, and to be treated exactly as if it were part of the original source file. All inserted source is flagged with an "@" on the listing. Only one level of .INSERT is allowed, they cannot be nested.

This pseudo-op will generate an "F" error if the file is not found, incorrectly specified, or if an .INSERT is already in progress.

Appendix G

Assembler Operation with CP/M

The TDL Z80 Relocating/Linking Assembler is initiated by the CP/M command:

```
ASM {sd:}file{.ext} {dd:}{switches}
```

where

- sd is the optional CP/M disk specification for the source file (defaults to the logged in disk)
- file is the source file name
- ext is the optional source file extension (defaults to ASM)
- dd is the optional CP/M disk specification for the output files (defaults to the same as the source file)
- switches are the optional assembly control switches, each of which is a single letter and which may appear in any order (with no intervening spaces)

The object file created by the assembly will have the same name as the source file, with an extension of .HEX if the .PABS option was used, and .REL if the .PREL option was used (the default).

Switches

A	.LALL
B	listing to both disk and list device
C	.LCTL
D	listing to disk (file name same as source with extension of PRN)
H	.PHEX (CP/M default is .PBIN)
I	.LIMAGE
K	.XLINK (CP/M default is .LINK)
L	listing only - no object file generated
O	object only - no listing generated
P	.PSYM
S	.SALL
X	.XLIST
Y	.XSYM

Note that all switches with pseudo-op equivalents will be overridden by contrary pseudo-ops within the source program.

Assembly Time Control

All of the assembly time control options (ctl-C, ctl-S, ctl-T) and page width options described in Appendix C also apply to the CP/M based version.

XITAN ZBUG : Z80 Debugger

User's Manual

June 12, 1978

Written by

Sidney R. Maxwell III

Table of Contents

XITAN Z80 ZBUG Debugger User's Manual
Table of Contents

Section 1

Introduction to ZBUG

ZBUG is XITAN's dynamic debugging utility, designed to facilitate assembly language programming. ZBUG, when used with XITAN's Macro Assembler, provides a powerful and versatile set of techniques for developing assembly language programs.

ZBUG provides standard debugging tools including memory and register examination/modification and program execution with breakpoints. However, ZBUG extends these common tools considerably with user-controlled data formatting, powerful expression evaluation for user-entered data, and extremely flexible trap capability with tracing.

This manual is intended as a guide for the user of ZBUG, beginning and experienced. For this reason, the sections of the manual are organized for general ease of access to basic and specific information.

Section 2 is provided as a guide to the first-time user, describing the basic features of ZBUG in a very general manner. It is not intended as a complete description, by any means, but is intended to give the flavor of ZBUG's possibilities.

Section 3 helps to demonstrate the use of ZBUG, indicating how to start execution of ZBUG, and giving a sample session to show the use of some of the basic commands. This section is intended to show the user how to begin using ZBUG as a useful tool.

Section 4 is a reference guide to the commands available in ZBUG, offering a complete description of each command's use and operation.

Section 5 is primarily concerned with giving the more experienced user an idea of the flexibility of the ZBUG command set. Included is a detailed discussion of the arithmetic expression capabilities of ZBUG for data type-in, and an advanced "session" with ZBUG to demonstrate ZBUG's muscle in handling different debugging situations. It is hoped that this section will lead to a more intimate understanding of ZBUG's abilities.

Finally, two appendixes are provided: a quick reference guide to the ZBUG command set, and a list of ZBUG's error messages.

This manual also documents UZBUG, a subset of ZBUG which occupies less memory. Notes throughout the manual define the ZBUG features not available in UZBUG.

Please note that this manual assumes that the ZBUG user is familiar with the Zilog Z80 CPU's register and flag organization, the Z80 instruction set, the Xitan Macro Assembler (the Z80 instruction mnemonics), and the Digital Research CP/M Disk Operating System. References are made to the following publications:

"Z80-CPU Technical Manual"
Zilog
170 State Street
Los Altos, California 94022

"An Introduction to CP/M Features and Facilities"
"CP/M Interface Guide"
Digital Research
Post Office Box 579
Pacific Grove, California 93950

"Z-80 Relocating/Linking Assembler User's Manual"
Xitan, Inc.
Research Park, Bldg. H
1101 State Road
Princeton, New Jersey 08540

Section 2

Overview of ZBUG

The following is a discussion of the main features of ZBUG's operation. It is intended to be a general introduction to the capabilities of ZBUG and a description of its use. For a complete description of each command available, please refer to Section 4.

The description of ZBUG is broken up into four general categories: data format, data examination and modification, execution and breakpoints, and tracing and traps. Each section mentions the commands offered in the particular category, and makes reference to the appropriate sub-section(s) of Section 4.

2.1 Data Format

Data can be interpreted in many ways. The length of the data (in 8 bit bytes for the Z80), its numeric representation, and whether considered as instructions or not are each important to the proper interpretation of data. ZBUG provides means for the user to interpret or specify data in different ways.

2.1.1 Data MODE

In ZBUG, memory data can be considered as a list of "cells" of a fixed length of one to four bytes, or as a list of Z80 instructions, each of varying length of one to four bytes.

In order to specify the manner or MODE memory data is to be displayed or accepted by ZBUG, the user employs the "M" command (see Section 4.8). This command sets the mode in which memory data is to be displayed/accepted, until overridden (see Sections 4.8 and 5.2) temporarily or the "M" command is used again to change the mode to another "default".

The modes that may be specified by the "M" command are byte, word (two byte), three byte, double word (four byte), and instruction. With all but instruction mode, data displayed and accepted is numeric, while in instruction mode, it is in XITAN Z80 instruction mnemonics (refer to the XITAN"Z80 Relocating/Linking Assembler User's Manual) with numeric operands as applicable.

Note that MODE and the "M" command are associated only with memory data. As registers and flags are of a fixed length, they may be considered as having a fixed mode - byte or word (two byte) for registers, and bit for flags.

2.1.2 Data RADIX

Numeric data, regardless of the mode in which it is considered, must have an understood RADIX in order to be interpreted properly. ZBUG provides the means to specify the radix for three different types of data - contents of memory "cells" or registers displayed by ZBUG, any numeric data typed by the user, and addresses of memory "cells" as displayed by ZBUG. The radix of each type of data may be separately specified by the user via the "R" command (see Section 4.12) and may be ASCII (with the exception of the address type), binary, decimal, hexadecimal, octal, split octal (3 digits per byte), and relative (signed) decimal.

The radixes specified by the "R" command determine the "default" used by ZBUG regarding the specific data types, and remain in effect until overridden (see Sections 4.12 and 5.2) temporarily or reset by another use of the "R" command.

UZBUG displays all data in hexadecimal, and therefore does not provide the "R" command. Data type-in may be in hexadecimal or ASCII (see Section 5.1).

2.1.2.1 Data Display

All numeric memory data (such as the contents of a "cell" in modes 1 (byte) through 4 (four byte), an instruction operand in mode instruction, or the contents of a register) are displayed by ZBUG in the radix set by the "RD" variation of the "R" command (see Section 4.12). This radix may be temporarily overridden in certain cases by a special application of the "R" command (see Sections 4.3, 4.12, 4.15, and 5.2), but always reverts to the radix last set by the "RD" command.

2.1.2.2 Data Type-in

All numeric data typed by the user is assumed to use the default radix set by the "RT" variation of the "R" command. This radix may be temporarily overridden with the use of the radix operator and/or modifier (see Sections 5.1, 5.2, and the XITAN "Z80 Relocating/Linking Assembler User's Manual), but always reverts to the radix last set by the "RT" command.

2.1.2.3 Address Display

Each address displayed by ZBUG, whether the address of a "cell", an instruction, or the address operand of an instruction, is displayed in the radix set by the "RA" variation of the "R" command. This radix may only be changed by the application of the "RA" command.

Addresses displayed by ZBUG are in one of two basic froms: absolute (the address displayed represents an actual physical address), and relative (the address displayed is a displacement relative to some absolute address).

ZBUG provides two pairs of "relocation registers" (see Sections 4.15 and 5.2) which are used by ZBUG to calculate relative addresses. The addresses contained in these registers (the RR - 'RR and DR - 'DR pairs) may be set to the beginning and ending addresses of any area of memory. If an address value lies between the beginning and ending addresses found in either pair of relocation registers, ZBUG will subtract the beginning address from the address to be displayed, forming a relative offset. This offset is then displayed, followed by a single quote (') if the offset is relative to the RR - 'RR pair, or a double quote (") if relative to the DR - 'DR pair.

If an address cannot be relocated to either base pair, or if ZBUG is commanded not to display relative addresses (via the "RA" command), the address is displayed as an absolute value.

UZBUG will always display addresses as absolute, and does not feature the RR or DR relocation register pairs.

If ZBUG finds that an address to be displayed lies between its own bounds, the address will be displayed as an offset relative to itself, followed by a pound sign (#).

2.2 Examining and Modifying Data

With the ability to specify the mode and radix data is to be displayed and accepted, the standard facilities of data examination and modification are greatly enhanced. ZBUG provides several commands facilitating manipulation of memory and register data.

2.2.1 Memory Data

ZBUG has six different commands for the manipulation of memory data - two for displaying data, two for replacing data, one for examination/modification of data, and one for searching for data.

2.2.1.1 Display Data

The "D" command (see Section 4.2) is used to display sequential "cells" (or instructions) in the current mode (as set by the "M" command) and radix (the "RD" command) along with their addresses (in the radix set by the "RA" command).

UZBUG displays both the data and their addresses in hexadecimal only.

The "L" command (see Section 4.7) is used to display ASCII printable data only, along with the associated address.

2.2.1.2 Replace Data

Using the "F" command (see Section 4.4), an area of memory may be filled with a numeric constant in the mode set by the "M" command (except for instruction mode - in this case, byte mode is used).

The "P" command (see Section 4.10) makes it easy to enter an ASCII string anywhere into memory.

The "O" command is used to load a CP/M format "COM" or "HEX" file (refer to Digital Research "An Introduction to CP/M Features and Facilities") or a XITAN "HEX" or "REL" file (refer to XITAN "Z80 Relocating/Linking Assembler User's Manual) for debugging purposes.

The "Z" command is used to reproduce the effects of entering a command string at the CP/M command level (see Section 4.17, and the Digital Research "CP/M Interface Guide"). With the aid of this command, the CP/M fcb's TFCB and TFCB+16, and the buffer TBUFF are set (or cleared) as defined in the "CP/M Interface Guide". This command is not available in UZBUG.

2.2.1.3 Examine/Modify Data

One of the most powerful commands in ZBUG's repertoire is the "E" command (see Section 4.3), which provides the means to examine and optionally modify memory. The "cell" is displayed, and modifying data accepted, in the current mode and data display radix, either of which may be overridden (see Sections 4.3, 5.1, and 5.2) temporarily. With this command, the "cell" currently under examination or opened may optionally be changed (merely by typing a replacement value or instruction), re-examined in a different mode/radix, or closed. Closing a "cell" can be followed the opening of the next sequential one, the last sequential one, a "called" one, a "returned from" one, or none - all with one keystroke (see Sections 4.3 and 5.2). With this command, a single memory cell or many may be examined and changed, a single instruction or a number of instructions examined and/or entered.

UZBUG will display both the data and addresses in hexadecimal always. A cell may be re-examined in a different mode, but not a different radix.

2.2.1.4 Search for Data

With the "Y" command (see Section 4.16), a string of "cells" or instructions (depending on the mode set by the "M" command) may be searched for in memory. ZBUG displays the addresses of each occurrence of the string found.

2.2.2 Registers and Flags

The "X" command (see Section 4.15) is provided to examine and optionally modify the contents of a machine register or flag (see Section 4.15, and the Zilog "Z80-CPU Tecnhical Manual") or a ZBUG psuedo register (see Sections 4.15, 5.1, and 5.2). The contents of a register are displayed in the radix set by the "RD" command, which can be overridden (see Sections 4.15 and 5.2) temporarily by special application of the "R" command. Flag values are always displayed as a 0 or 1.

UZBUG displays register values in hexadecimal, and the special application of the "R" command does not apply.

2.3 Program Execution and Breakpoints

ZBUG implements the standard "goto with breakpoints" in the form of the "G" command (see Section 4.5).

2.3.1 Executing a Program

With the "G" command, complete transfer of control from ZBUG to the address specified is made - i.e., the machine is no longer under ZBUG's supervision. Until a breakpoint is reached, the user's program has complete control at normal execution speed.

2.3.2 Breakpoints During Execution

The use of the "G" command includes the setting of up to seven individual software breakpoints by ZBUG before tranfer of control to the user program is effected. These breakpoints take the form of a "restart 6" instruction (RST 6 - or 0F8 hex), and ZBUG assumes the user program does not use this instruction and does not modify locations 030-032 hex in memory.

2.4 Tracing and Traps

Perhaps the most useful and powerful features of ZBUG are its tracing and trap capabilities.

2.4.1 Tracing a Program

Utilizing the "T" command (see Section 4.14), it is possible to execute a program while under ZBUG's full supervision. The user may specify the number of instructions to be traced. Unless interrupted (by a trap condition, invalid instruction, or user intervention), ZBUG will execute the program, simulating actual execution at a

speed of 250-2500 times slower (depending on various conditions).

While tracing, ZBUG may be instructed to display the instructions executing, and the values of certain machine registers modified by those instructions.

UZBUG will always display the executing instructions and modified machine registers.

2.4.2 Traps During Tracing

The "S" command (see Section 4.13), in combination with the trace capability of ZBUG, provides the ability to trace a program until any one of four arbitrary conditions occurs. With the "S" command, up to four boolean expressions (see Sections 5.1 and 5.2) may be saved. Each of the saved expressions are evaluated after each instruction traced. If any one of the evaluated expressions returns a non-zero value, ZBUG halts tracing, notifying the user of the trap.

As traps are determined and controlled by arbitrary boolean expressions, traps may be set to monitor register/flag/memory value compared to a constant or other register/flag/memory value(s). A register or flag may be compared to its own value prior to the last instruction execution, in order to monitor a change in value. It is possible to trap when a register/flag/memory value reaches a certain value (or range of values), or when a specific instruction is about to be executed.

The potential of this feature is limited only by the user's understanding and use of the conditional expressions that may be devised.

UZBUG does not provide the trap features found in ZBUG, and tracing interruption is left to user intervention or invalid instructions.

Section 3

Starting Out

This section is provided to introduce the user to ZBUG. Included in the discussion following is a description of the environment under which ZBUG runs, how to execute ZBUG in order to debug an existing program, and basic debugging operations using a few simple commands.

3.1 ZBUG's Operating Environment

ZBUG is designed to operate under the Digital Research CP/M Operating System. The files DEBUG.COM and ZBUG.REL must be present on the currently logged-on drive (or drive A) in order to run ZBUG.

The program DEBUG.COM is initiated at the CP/M command level (refer to the Digital Research "An Introduction to CP/M Features and Facilities"), which loads ZBUG from the ZBUG.REL file. ZBUG is loaded, following CP/M convention, as high in memory as allowed by the operating system, leaving any low memory free to contain the program to be debugged.

All ZBUG disk and console I/O is performed using CP/M Interface Guide").

ZBUG currently occupies approximately 13.25 K bytes, including all data areas.

UZBUG operates in the same manner as ZBUG, with the files UDEBUG.COM and UZBUG.REL replacing DEBUG.COM and ZBUG.REL of ZBUG.

UZBUG occupies approximately 9.25 K bytes.

3.2 Executing ZBUG

There are two methods of invoking ZBUG....

While at the CP/M command level, the command (in lower case):

```
A> debug <cr>
```

will cause the file DEBUG.COM to be loaded and executed. DEBUG will display ZBUG's current size (in bytes) and load address (which indicates the size of free memory), in hex. DEBUG then will load the ZBUG.REL file, relocating it to the highest available memory address. If ZBUG.REL is not located on the currently logged-in drive, DEBUG will look for it on drive A. DEBUG will finally pass control to ZBUG, which will display a sign-on message and prompt, after which the user may begin entering commands to ZBUG.

The alternate method of executing ZBUG is as follows. Again at the CP/M command level, the command:

```
A> debug <name> <cr>
```

will cause DEBUG to load the file <name>.COM at location 100 hex (after signaling a successful load of ZBUG), and display the size (in bytes), the load address, and the end address, before passing control to ZBUG. This action is the same as executing ZBUG in the previous manner, and immediately entering the following command to ZBUG:

```
* o <name> <cr>
```

Please note that in the examples above, <name> is a CP/M filename (i.e., [device :] name [. extension]). With the ZBUG "O" command, a file with the extension "COM", "HEX", or "REL" may be loaded, as described in Section 4.9. If DEBUG loads the file, however, only files with the "COM" extension are loaded. DEBUG ignores any extension entered at the CP/M command level, changing it to "COM".

If DEBUG encounters an error while loading either ZBUG or the optionally specified "COM" file, a short error message will be displayed, and control returned to the operating system.

3.3 A Sample Session

This following is a sample workout with ZBUG. A few basic commands will be explored, with the idea of presenting ZBUG's usefulness without getting bogged down with the more involved features.

For this session, assume that the following is a listing of an assembled program which exists on the logged-in drive as the "COM" file PRINT.COM:

```
                        ;
                        ;    A program to list the 10 bytes
                        ;    starting at location 80 hex on the
                        ;    console in hex
                        ;
                             .pabs
                             .phex
0100                    .loc  ^H100
0100  21 0080   print:  lxi   h,^H80   ;start at ^H80
0103  060A              mvi   b,10     ;   and list 10 bytes
0105  7E        loop:   mov   a,m      ;get a byte
0106  CD 010F           call  hex      ;   and list in hex
0109  23                inx   h        ;bump to next byte
010A  10F9              djnz  loop     ;   and loop if more
010C  C3 0000           jmp   0        ;leave when done
010F  F5        hex:    push  psw      ;save byte
0110  1F                rar            ;rotate
0111  1F                rar            ;   to get
0112  1F                rar            ;      high
0113  1F                rar            ;         nibble
0114  CD 011F           call  nibble   ;print high nibble
0117  F1                pop   psw      ;restore byte and
0118  CD 011F           call  nibble   ;   print low nibble
011B  3E20              mvi   a,' '    ;print a space
011D  1804              jmpr  output   ;   and return
011F  E60F      nibble: ani   ^H0F     ;mask out nibble
0121  C630              adi   '0'      ;   and add ASCII zero
0123  C5        output: push  b        ;save <BC
0124  E5                push  h        ;   and <HL
0125  5F                mov   e,a      ;CP/M convention
0126  0E02              mvi   c,2      ;   to print character
0128  CD 0005           call  5        ;      on console
012B  E1                pop   h        ;restore <HL
012C  C1                pop   b        ;   and <BC
012D  C9                ret            ;return
0100                    .end  print
```

The following is a verbatim listing of a terminal session. The user's input is in lower case alphabetics, the computer's response in uppercase. Comments of the related user and computer actions are enclosed in curly brackets ("{ ... }"). The constructions "<cr>" and "<lf>" in the session are user-typed carriage return and line feed, respectively.

{ Starting at the CP/M command level, we'll try executing PRINT, to see what happens.... }

```
A>print <cr>
00 00 42 55 47 2= 34 20 20 44
A>
```

{ Obviously, something appears to be wrong - where did the "2=" come from ? }

```
A>debug print <cr>

ZBUG's length - ^H0351F and
load address  - ^H03AE1

ZBUG is loaded....
loading COM file....
^H00200 bytes loaded -- (from ^H00100 to ^H002FF)


XITAN Z80 CP/M DYNAMIC DEBUGGER VERSION 1.00

*
```

{ First, let's take a look at location 80 hex to see the 10 bytes we're supposed to be listing.... }

```
* e 80 <cr>
0080:   10      <lf>
0081:   D8      <lf>
0082:   C3      <lf>
0083:   2A      <lf>
0084:   05      <lf>
0085:   CD      <lf>
0086:   41      <lf>
0087:   00      <lf>
0088:   CA      <lf>
0089:   2A      <lf>
008A:   05      <cr>
```

{ Okay, now let's execute it to see the results, but with a breakpoint at the jump back to CP/M }

```
 * g 100, 10c <cr>
 10 =8 <3 2: 05 <= 41 00 <: 2: *** BREAK (0) --> 010C: JMP
0000
```

{ Yes, folks - something is definately amiss. For each hex digit greater than 9 (as near as we can tell so far), we have a garbage character - ":" for A, "<" for C, and "=" for

D. Let's start it up again, stopping after getting the second byte (the first with a problem) from memory. }

```
* g 100, 10f <cr>
*** BREAK (0) --> 010F: PUSH    PSW
```

{ That was the first, loop for the second.... }

```
* g, 10f <cr>
10 *** BREAK (0) --> 010F:      PUSH    PSW
```

{ Okay, now look at that byte, in the A register }

```
* xa <cr>
D8      <cr>
```

{ Yes, it's the right byte, alright. Now go until we've isolated the top nibble - the one with the problems... }

```
* g, 11f <cr>
*** BREAK (0) --> 011F: ANI     0F
```

{ ... and look again. }

```
* xa <cr>
0D      <cr>
```

{ Right - the nibble in set up properly. So, we'll go ahead and mask it and add the '0'... }

```
* g, 123 <cr>
*** BREAK (0) --> 0123: PUSH    B
```

{ ... and look at the result - in ASCII since we're ready to print out what's left. Note that in UZBUG we can't do this - we'd have to stay in hex }

```
* xa <cr>
3D      ;ra <cr>
'='     <cr>
```

{ There's the problem. We now remember that a 0D hex added to a '0' (30 hex) is not the 'D' (44 hex) we wanted. All we have to do is add a check to see if the character we're going to print is greater than a '9' (39 hex), and add enough to get the proper character. We'll need to know what that extra value is so.... }

```
* c 'D'-3d <cr>
07
```

{ We see that to get a 'D' instead of a '=', we have to add a 7. Now, we'll modify the program to include the check and add. First, we should take advantage of ZBUG's assembler/disassembler capabilities, and get into instruction mode. }

```
    * mi <cr>
    MODE: INSTRUCTION

    { Now, let's change that "nibble" routine.... }

    * e 11f <cr>
    011F:   ANI     0F      <lf>
    0121:   ADI     30      <lf>

    { We'll put the check right on top of where the "output"
routine is, making it garbage.... }

    0123:   PUSH    B       cpi     '9'+1 <lf>
    0125:   MOV     E,A     jrc     .+4 <lf>
    0127:   STAX    B       adi     7 <lf>

    { Now we must relocate the "output" routine.... }

    0129:   DCR     B       push    b <lf>
    012A:   NOP             push    h <lf>
    012B:   POP     H       mov     e,a <lf>
    012C:   POP     B       mvi     c,2 <lf>
    012E:   JRNZ    0150    call    5 <lf>
    0131:   MOV     B,H     pop     h <lf>
    0132:   MOV     D,H     pop     b <lf>
    0133:   JRNZ    018B    ret <cr>
    *

    { Okay, now to fix up the relative jump to "output"... }

    * e 11d <cr>
    011D:   JMPR    0123    jmpr    129 <cr>

    { ... and try it out, stopping again before running off
to CP/M. }
 * g 100, 10c <cr>
 10 D8 C3 2A 05 CD 41 00 CA 2A *** BREAK (0) --> 010C: JMP
0000
 *
    { Fine, all's well.  We'll leave ZBUG... }

    * q <cr>

    A>

    { ... and save it. }

    A>save 2 print.com <cr>
    A>

    { One last time.... }

    A>print
    00 00 42 55 47 2D 34 20 20 44

    { All fixed ! }
```

Section 4

The Commands - A Detailed Description

The following is a listing of the commands ZBUG provides, with a detailed description of the use and operation of each. This section is intended for use as a detailed reference guide to ZBUG.

The format of each command is as follows:

Command character - Command name
Command format
Description
Example

The following symbols are used to describe the format to the various ZBUG commands:

<CR> represents a carriage return,
<LF> a line feed,
<BS> a backspace (control-H),
<ESC> an escape (or altmode)
<FF> a form feed (control-L)

[...] means contents are optional,
{ ... } means contents are mandatory,
... | ... means "or" - i.e., a choice can be made
...]+ -or- ...}+ means one or more
...]* -or- ...}* means zero or more

Please note that ZBUG is only blank (' ') sensitive where expressly stated. Normally, blanks may be used freely during type-in to facilitate easier reading. Note also that more than one command may be typed on the same line, separated by semicolons (';') and terminated by a <CR>. ZBUG will attempt to execute each command in order of appearance, unless a) an error occurs, or b) an "E", "X", or "Y" command has been executed, after which any remaining commands will be ignored.

Unless explicitly defined otherwise, all constructions of the form "< ... >" in the command descriptions (such as <expression>, <address>, <count>, etc.) are properly formed ZBUG expressions (see Section 5.1).

In each of the command examples, the ZBUG prompt ("*") is followed by the user input (in lower case alphabetics), and then by any ZBUG response. Unless otherwise stated, the mode is byte and the radixes (address display, data display, and default type-in) are hexadecimal.

4.1 C - Calculate

C <expression>

Calculate the value of <expression> and display in the current mode and data display radix.

The <expression> is evaluated and its value displayed in the current mode and data display radix. If the <expression> is omitted, no value is displayed.

UZBUG will display the resulting value in hexadecimal.

```
{ calculate 1+2 }

* c 1+2 <cr>
03
*

{ calculate 3*5 }

* c 3*5 <cr>
0F
*
```

4.2 D - Display

D [<address>] [, <count>]

Display memory in the current mode and data display radix.

Starting at <address>, display <count> sequential cell addresses (in the current address display radix) and cell contents (in the current mode and data display radix). If <address> is omitted, the default is 0. If <count> is omitted, the default is 1. ZBUG formats the display, placing an address and one, two, four, or eight values per line (depending on mode and data display radix).

UZBUG will display both the data and addresses in hexadecimal.

```
{ display cell at 0 }

* d <cr>
0000:   F8
*

{ display 5 cells starting at 100 }

* d 100, 5 <cr>
0100:   37 23 40 F7 EA
*

{ set instruction mode (see Section 4.8), then
display 4 cells (instructions) starting at 200 }
```

```
* mi <cr>
MODE:  INSTRUCTION
* d 200, 4 <cr>
0200:   ANI     04
0202:   JRNZ    2010
0204:   POP     PSW
0205:   RET
*
```

4.3 E - Examine

E [<address>]

Open cell at <address> for examination and modification.

Open the cell at <address>, displaying <address> in the current address radix, and the contents of the cell in the current mode and data display radix. Accept from the user an optional replacement value - assumed to be in the current mode - followed by a valid closing character. Note that if the current mode is instruction, the user may type an instruction (mnemonic/operand(s) sequence). When accepting an instruction as a replacement value, ZBUG is blank sensitive, as a blank or tab MUST separate a mnemonic from any operand(s) required by the particular instruction.

UZBUG will display both the cell data and address in hexadecimal.

After the replacement value has been typed, or instead of it, the user must close the location to continue on. To close the location, he/she may do one of the following:

type a <CR> to close and exit the E command,

type a <LF> to close and open the next sequential cell,

type a comma (',') to close and open the next sequential cell on the same line (not valid in instruction mode, as a ',' separates instruction operands),

type a <BS> to close and open the last sequential cell (in instruction mode, the mode is first changed to byte (see * Note)),

type a <ESC> to close and open the cell pointed to by the last value (or address, if instruction mode) typed or displayed, pushing the "return" address of the next sequential cell on the "call" stack,

type a <FF> to close, popping the "return" address from the "call" stack and opening that cell,

type a semicolon (";") followed optionally by "Mn" (see Section 4.8) and/or "Rn" (see Section 4.12), followed by a <CR>, to optionally change the current mode and/or data display radix temporarily (see * Note) and reopen the cell.

(Note that the "Rn" option is not available in UZBUG)

Note that, for each newly opened cell, ZBUG goes to a new line, displays the cell's address (in the current address display radix) and the cell's contents (in the current mode and data display radix).

* Note: The changing of the mode and/or data display radix as indicated above changes same only until a) it is changed again in the same manner or b) a cell is closed with a <CR>, exiting the E command and restoring the mode/radix of before. Note also that the radix cannot be changed in UZBUG.

Note that the special value "." ("here") always contains the address of the cell currently open, and on exit from the E command, contains the address of the last cell opened. For more information regarding ".", refer to Section 5.1.

```
{ examine cell at 100 }

* e 100 <cr>
0100:   C3      <cr>
*

{ examine and modify cells... }

* e 100 <cr>
0100:   C3      <lf>
0101:   00      05 <lf>
0102:   34      <cr>
*

{ set instruction mode, and try a few things... }

* mi <cr>
MODE:  INSTRUCTION
* e 100 <cr>
0100:   JMP     3405    <lf>
0103:   CALL    2156    <lf>
0106:   JRNZ    0113    <esc>
0113:   CPI     41      ;ra <cr>
0113:   CPI     'A'     <lf>
0115:   JRZ     0134    <ff>
0108:   ANI     '^A'    ;rh <cr>
0108:   ANI     01      ANI     02 ; <cr>
0108:   ANI     02      <cr>
*
```

Please refer to Section 5.2 for more discussion.

4.4 F - Fill

F [<address>] , [<count>] , <value>

Fill cells with constant.

Starting with <address>, fill <count> sequential cells in the current mode (if instruction mode, assume byte) with the value <value>. If <address> is omitted, default to 0. If <count> is omitted, default to 1.

{ fill starting at 100 with 4 '!''s }

* f 100, 4, '!' <cr>
*

{ put a 0 in cell I00 }

* f 100, , 0 <cr>
*

4.5 G - Goto

G [[<start>] [, <break>]*]

Goto <start> with breakpoints at <break>.

Clear any previously set breakpoints. For each <break> address (ZBUG provides up to seven) indicated, set a software breakpoint. Begin execution (complete transfer of control) at the address <start>. If a breakpoint is encountered, interrupt execution and notify the user of the break, indicating which trap occurred (0 - 7) and displaying the address and associated instruction. If no breakpoints were specified, do not set any. If <start> was omitted, default to the address in the program counter (register <PC).

ZBUG utilizes a "restart 6" (RST 6 -or- 0F8 hex) instruction for software breakpoints, which require that ZBUG place a jump instruction at locations 0030 to 0032 hex for proper operation of breakpoints. This implies that the user program must NOT use this instruction or modify these locations, or undefined actions may result.

When a break has been encountered and the user notified, ZBUG places the address of the break (kept in the program counter - register <PC) in the special value ".". For more information, refer to Section 5.1.

{ start execution at 100 }

* g 100 <cr>

{ start execution at the address in the program counter, with a breakpoint at 340 }

```
* g, 340 <cr>
*** BREAK (0) --> 0340: CALL    2351
*
```

4.6 I - Instruction Interpret

I <instruction>

Interpretively execute the <instruction>.

After "assembling" the <instruction>, interpretively execute it.

This command provides the capability of executing an arbitrary instruction, in order to effect the actions determined by the particular instruction. Unless the instruction performs a transfer of control (i.e., jumps, calls, returns, restarts, or indirect register jumps), the program counter will not be changed.

ZBUG effectively traces the one instruction and returns to the user.

Note that if the <instruction> is a call or restart instruction, the associated return address pushed on the stack will be within ZBUG which, when transfered to by a matching return (or any other means), will cause ZBUG to notify a breakpoint, and return to the user.

This command is not available in UZBUG.

{ execute an increment register A instruction checking the contents before and after }

```
* x a <cr>
03      <cr>
* i inr a <cr>
* x a <cr>
04      <cr>
*
```

{ execute a push register pair H&L instruction and check the stack pointer }

```
* x sp <cr>
0546    <cr>
* i push h <cr>
* x sp <cr>
0544    <cr>
*
```

4.7 L - List ASCII

L [<address>] [, <count>]

Starting at <address>, list <count> ASCII characters.

Starting at <address>, display <count> sequential printable ASCII charaters (if nonprinting, print a "."), preceding each group of up to 32 characters with the address (in the current address display radix) associated. If <address> is omitted, default to 0. If <count> is omitted, default to 1.

ZBUG formats the display, so that for every line displayed contains one address and up to 32 ASCII characters.

UZBUG will display the address in hexadecimal.

```
{ list the 23 characters starting at 100 }

* l 100, 23. <cr>
0100:   ...A.+;@....%0Aa*:....j
*

{ list the character at 0 }

* l <cr>
0000:   .
*
```

4.8 M - Mode

M [<modifier>]

where <modifier> is { B | W | I | 1 | 2 | 3 | 4 }

Set current mode.

Set the current mode to byte ("B" or "1"), word or double byte ("W" or "2"), triple byte ("3"), double word or four byte ("4"), or instruction ("I"), and display a message reflecting the change. If the mode modifier is omitted, display the current setting.

A special application of this command is available during the execution of the "E" (examine) command. In place of the replacement value accepted by ZBUG (or immediately following it), the user may type: ";Mn", where "n" is one of the modifiers described above. This changes the current mode for the remainder of the execution of the command, unless changed by another application of this feature.

When ZBUG is first executed, the mode is set to byte.

```
{ change mode to instruction }
```

```
* m i <cr>
MODE:  INSTRUCTION
*

{ change mode to word (two byte) }

* m 2 <cr>
MODE:  WORD
*

{ display the current mode setting }

* m <cr>
MODE:  BYTE
*
```

4.9 O - Open File

O <filename> [, <bias>] [, <relocation>]

Open <filename> for debugging.

Load CP/M disk file <filename> into memory, with optional bias of <bias> and relocation (if a XITAN ".REL" file) of <relocation>. If <bias> is omitted, default to 0. If <relocation> is omitted, default to 100 hex.

<filename> is a CP/M filename of the form:

[<drive> :] <name> [. <extension>]

where the extension is "COM" for a binary image file, "HEX" for an Intel-compatible "hex" object file (whether binary or ASCII), and "REL" or a XITAN relocatable object file (binary or ASCII). If the extension is omitted or not "COM", "HEX", or "REL", a "COM" type file is assumed.

Files are loaded to the actual physical addresses found as follows:

COM files:	The <bias> (defaulting to 0 if omitted) plus 100 hex.
HEX files:	The <bias> (defaulting to 0 if omitted) plus the load address supplied in the HEX format.
REL files:	The <bias> (defaulting to 0 if omitted) plus the <relocation> (defaulting to 100 hex if omitted).

If the file <filename> cannot be found, or if an error has occurred while reading it, or the file attempts to be loaded within ZBUG's bounds, an error will result, and ZBUG will discontinue loading.

During the loading, ZBUG displays the starting load address and the ending load address. If an error is encountered in a "HEX" or "REL" file's format while loading, the address of the last byte loaded will be displayed.

Note that ZBUG is blank sensitive within a filename, assuming any blank encountered terminates the filename.

```
{ open COM file TEST, loading it starting at 100 }

* o test <cr>
0100:   LOAD ADDR
05FF:   END ADDR
*

{ open file TEST.REL, loading and relocating it at 200 }

* o test.rel, , 200 <cr>
0200:   LOAD ADDR
0342:   END ADDR
*

{ open COM file FILE from drive A }

* o a:file <cr>
0100:   LOAD ADDR
037F:   END ADDR
*
```

4.10 P - Put String

P [<address>]

Put an ASCII string into memory starting at <address>

After displaying the <address> in the current address display radix, accept ASCII charaters from the user, storing them in sequential memory bytes starting at <address>, until a control-D is typed. If <address> is omitted, default to 0.

UZBUG will display the address in hexadecimal.

```
{ put a string at 100 }

* p 100 <cr>
0100:   this is a string <control-D>
*

{ put a string at 0 }

* p <cr>
0000:   example <control-D>
*
```

4.11 Q - Quit

```
Q

Exit ZBUG, returning to CP/M.

{ quit and go to CP/M }

* q <cr>
A>
```

4.12 R - Radix

```
R [ <type> [ <modifier> ]]

where <type> is { A | D | T }
and <modifier> is { A | B | D | H | O | R | S }
```

Set current address display, data display, default type-in radix.

Set the current address display (<type> "A"), data display (<type> "D"), or default type-in (<type> "T") radix to either ASCII ("A"), binary ("B"), decimal ("D"), hexadecimal ("H"), octal ("O"), relative or signed decimal ("R"), or split octal ("S"), and display a message reflecting the change. If the radix modifier is omitted, display the current setting for the radix type <type>. If the radix type is omitted, display the current settings of each radix type.

A special application of this command is available during the execution of the E (examine) and X (examine register/flag) commands (see Sections 4.3 and 4.15). In place of the replacement value accepted by ZBUG (or immediately following it), the user may type: ";Rn", where "n" is one of the radix modifiers described above. This changes the data display radix for the remainder of the execution of the command, unless changed by another application of this feature.

Note that the ASCII ("A") radix is not valid for the address display radix type.

An additional flexibility is provided for the address radix application. If the radix is being modified, and the command is followed by an "A" (i.e., "RAHA"), ZBUG recognizes that addresses will be displayed as absolute values, and not relocated (see Sections 2.1.2.3, 4.15, and 5.2).

When ZBUG is first executed, each of the radix types are set to hexadecimal, and relative addresses are permitted (i.e., the "A" option for the address radix is not in effect).

This command, and all its variations, are not available in UZBUG.

```
{ change data display radix to hexadecimal }

* r dh <cr>
DATA DISPLAY RADIX : HEXADECIMAL
*

{ display the current address display radix setting }

* r a <cr>
ADDRESS DISPLAY RADIX : BINARY
*

{ set the address radix to hexadecimal, specifying
absolute addresses only }

* r aha <cr>
ADDRESS DISPLAY RADIX : (ABSOLUTE) HEXADECIMAL
*

{ display the current settings of each of the radix
types }

* r <cr>
ADDRESS DISPLAY RADIX : BINARY
DATA DISPLAY RADIX : HEXADECIMAL
DEFAULT TYPE-IN RADIX : DECIMAL
*
```

4.13 S - Set Trap/Conditional-Display

```
S [ D ] [ * ] [ <id> [ , <expression> ]]

where <id> is { 0 | 1 | 2 | 3 }
```

Set trap/conditional display.

This command performs several different functions, depending on the options used. Each is described separately below.

```
S [ <id> [ , <expression> ]]
```

Set trap <id> to boolean expression <expression>. If the <expression> is omitted, display the expression set currently for trap <id>. If <id> is omitted, display the expressions currently set for each trap.

```
S* [ <id> ]
```

Reset trap <id>, removing its currently set expression, and effectively clearing the trap. If <id> is omitted, display the id's of each cleared trap.

SD [<id> [, <expression>]]

Set conditional-display <id> to boolean expression <expression>. If the <expression> is omitted, display the expression saved currently for conditional-display <id>. If <id> is omitted, display the expressions currently set for each conditional-display.

SD* [<id>]

Reset conditional-display <id>, removing its currently set expression, and effectively clearing the conditional-display. If <id> is omitted, display the id's of each cleared conditional-display.

For each trap or conditional-display set, ZBUG saves the <expression> specified, after surrounding it with parentheses and preceding the resulting expression with a unary radix change operator for the current default type-in radix. This is to insure the user that the expression will be evaluated during tracing in the default radix active when the expression was originally entered, regardless of any subsequent later changes with the R command. Each expression entered, whether trap or conditional-display, must be no more than 59 characters long, including any spaces or tabs contained within.

During tracing (see Section 4.14), each expression saved is evaluated after every instruction traced. The effects of and uses of both types of expression (trap and conditional-display) are described further in Section 4.14 and 5.2.

This cammand, and all its variations, is not available in UZBUG.

```
{ set trap 0 to fire when register A changes }

* s0,<a ?ne <!a <cr>
*

{ set trap 3 to fire when the next instruction to be
traced is a pop register pair D&E }

* s3, !. ?eq [pop d] <cr>
*

{ display all currently set traps }

* s <cr>
(0)     ^H(<A ?NE <!A)
(3)     ^H( !. ?EQ [POP D])
*

{ display all currently cleared trap id's }

* s* <cr>
1 2
*
```

{ set conditional-display 0 to display if the program counter is between 1000 and 1296 }

```
* sd0,(<pc ?ge 1000) & (<pc ?le 1296) <cr>
*

i{ display currently set conditional-displays }

* sd <cr>
(0)     H((<PC ?GE 1000) & (<PC ?LE 1296))
*

{ display currently cleared conditional-displays }

* sd <cr>
1 2 3
*
```

4.13.1 SW - Set Wait

SW [<count>]

Set a delay time of <count> * 10 msec (at 2 MHZ) for tracing.

Set tracing delay time of <count> (default type-in radix is alway decimal for this command) centi-seconds. If <count> is omitted, display the current setting in decimal centi-seconds.

During tracing, ZBUG will wait for the count set by this command after displaying an instruction and before executing it. This gives the user time to see what is about to be executed, and interrupt ZBUG before anything happens if desired. The count is considered by ZBUG to be between 0 and 255 centi-seconds, giving the user up to just over 2.5 seconds to make the decision to interrupt, or enough time to follow the trace with a listing.

{ set delay to maximum }

```
* sw 255 <cr>
*

{ display setting }

* sw <cr>
255
*
```

4.14 T - Trace

T [<address>] [, <count>] [, { O | C }]

Trace <count> instructions starting at <address>.

Starting at <address>, trace up to <count> instructions. If <count> is omitted, default to 1. If <address> is omitted, start at the address in the program counter.

If the "O" or "C" options are omitted, the automatic display is in effect. This implies that before the execution of each instruction, ZBUG will display the address of the next instruction to be executed (in the current address display radix) and the instruction (with any operands in the appropriate current address or data display radix). ZBUG will not go to a new line (the cursor will remain on the same line as the instruction) until the instruction displayed is executed.

After the instruction has been executed, display the contents of the SP, IX, IY, AF, BC, DE, and/or HL registers if they were modified by the instruction, and then display the next instruction.

If the "O" option is used, turn off automatic display.

If the "C" option is used, turn off automatic display only when tracing a subroutine (code entered by a call instruction and left by a matching return instruction) in a deeper level (up to 128 calls deep). With this option, the <count> refers only to instructions that are displayed.

Before executing each instruction, evaluate each currently set conditional-display expression. If a non zero value is found, display the id (of the expression causing it) and the instruction to be executed. If no expression value is non-zero, display without an id if the automatic display is in effect, otherwise do not display.

If the current instruction has been displayed (by the automatic display and/or a conditional-display), wait for the time specified by the SW command before executing it and going to a new line. The pause before new-line gives the user time to interrupt tracing by the use of a control-E.

Before executing each instruction, save the present value of all machine registers and flags as the next "old" values.

After executing each instruction, set the special value "." ("here" - see Section 5.1) equal to the value in the program counter.

After executing each instruction, evaluate each currently set trap expression. If any non-zero values are found, display the related id's and expressions, the next instruction, and stop tracing.

If, during tracing, the user types a control-E, or a halt (HLT) instruction or invalid instruction is encountered, halt tracing at the current location.

If, during tracing, the user types a control-T, display the current instuction being traced.

Note that the <count> specified by the user in this command is treated by ZBUG to be a positive 16-bit value, with a default of 1. If it is desired that ZBUG trace indefinitely, a count of 0 will result in an infinite trace.

UZBUG does not provide the "O" or "C" options, traps, conditional display, or the control-T features available in ZBUG.

```
{ trace the next instruction }

* t <cr>
        0238:   INR     A       - AF (1501)
*
```

```
{ assuming the trap and conditional-display settings of
the S command examples of Section 4.13, begin an "infinite"
trace to see what happens }

* t 100, 0 <cr>
        0100:   ORA     A       - AF (1500)
        0101:   JNZ     1200
(0)     1200:   MVI     B,21    - BC (2100)
(0)     1202:   LXI     H,0000  - HL (0000)
(0)     1205:   CCIR            - AF (0012) - BC (0000)
- HL (2100)
(0)     1207:   JNZ     1450
        1450:   PUSH    D       - SP (050C)
        1451:   MOV     A,B     - AF (0012)
*** TRAP (0) --> ^H(<A ?NE <!A)
1452:   ORA     A
*

{ ... and going on from there... }

* t, 0 <cr>
        1452:   ORA     A       - AF (0044)
        1454:   JRZ     146A
*** TRAP (3) --> ^H( !. ?EQ [POP D])
146A:   POP     D
*

{ now, do it all again, but this time, turn off the
automatic display off }

* t 100, 0, o <cr>
(0)     1200:   MVI     B,21    - BC (2100)
(0)     1202:   LXI     H,0000  - HL (0000)
(0)     1205:   CCIR            - AF (0012) - BC (0000)
- HL (2100)
(0)     1207:   JNZ     1450
```

```
*** TRAP (0) --> ^H(<A ?NE <!A)
1452:   ORA     A
*
```

{ start a trace, assuming trap 0 is set to fire when the program counter is between 145 and 14E }

```
* t 100, 0
        0100:   MVI     A,01        - AF (0100)
        0102:   LXI     D,0000      - DE (0000)
        0105:   CALL    1230        - SP (05F0)
        1230:   PUSH    H           - SP (05EE)
        1231:   PUSH    B           - SP (05EC)
        1232:   MOV     B,A         - BC (010E)
        1233:   XCHG                - DE (1257) - HL (0000)
        1234:   MVI     M,0
        1236:   DJNZ    1234        - BC (000E)
        1238:   POP     B           - SP (05EE) - BC (1537)
        1239:   POP     H           - SP (05F0) - HL (0012)
        123A:   RET                 - SP (05F2)
        0108:   JMP     0140
        0140:   LXI     H,0000      - HL (0000)
        0143:   LXI     B,FFFF      - BC (FFFF)
*** TRAP (0) --> ^H((<PC ?GE 145) & (<PC ?LE 14E))
0146:   XRA     A
*
```

{ now, do the same, but with the C option }

```
* t 100, 0, c
        0100:   MVI     A,01        - AF (0100)
        0102:   LXI     D,0000      - DE (0000)
        0105:   CALL    1230        - SP (05F2)
        0108:   JMP     0140
        0140:   LXI     H,0000      - HL (0000)
        0143:   LXI     B,FFFF      - BC (FFFF)
*** TRAP (0) --> ^H((<PC ?GE 145) & (<PC ?LE 14E))
0146:   XRA     A
*
```

4.15 X - Examine Register/Flag

X [[<] <register-name> | > <flag-name>]

where <register-name> is:

```
[ ! ] [ ' ]   { AF | BC | DE | HL |
                SP | PC | IX | IY |
                IR | RD | WR | RR |
                DR |
                A | F | B | C | D |
                E | H | L | I | R |
                M }
```

and <flag-name> is:

[!] ['] { C | H | N | P | S | V | Z }

Open register/flag for examination and modification.

Open the specified register or flag, displaying its contents in the current data display radix (or a 0 or 1 if display a flag). Accept from the user an optional replacement value followed by a valid closing character.

After the replacement value has been typed, or instead of it, the user must close the register or flag, by doing one of the following:

type a <CR> to close the register or flag and exit the X command;

type a semicolon (";") followed optionally by "Rn" (see Section 4.12), followed by a <CR>, to optionally change the current data display radix temporarily (see * Note) and reopen the register (note that this does not work if examining a flag).

* Note: The change of data display radix as indicated above is in effect only until a) it is changed again in the same manner as above or b) the register is closed with a <CR>, exiting the X command and restoring the previous mode.

If no register or flag name is specified, display the contents of all the machine registers, the pseudo registers, the top four word values on the stack, and the instruction pointed to by the program counter.

The register names defined above are further described below:

The 16-bit registers:

AF - The Z80 register pair commonly known as PSW
BC - The Z80 register pair B&C
DE - The Z80 register pair D&E
HL - The Z80 register pair H&L
SP - The Z80 stack pointer
PC - The Z80 program counter
IX - The Z80 index register X
IY - The Z80 index register Y
IR - The Z80 register "pair" made of the combining
- of the interrupt register and the refresh
- register
RD - The ZBUG pseudo register containing the
- address of the last traced memory read
- access
WR - The ZBUG pseudo register containing the
- address of the last traced memory write
- access
RR - The ZBUG pseudo register containing the
- user defined "code" relocation address
DR - The ZBUG pseudo register containing the
- user defined "data" relocation address

The 8-bit reisters:

```
A       - The Z80 register A (accumulator)
F       - The Z80 flag "register"
B       - The Z80 general register B
C       - The Z80 general register C
D       - The Z80 general register D
E       - The Z80 general register E
H       - The Z80 general register H
L       - The Z80 general register L
I       - The Z80 interrupt register I
R       - The Z80 refresh register R
M       - The Z80 "register" M (byte pointed
        - to by register pair H&L
```

The flag names described above are further defined below:

```
C       - The Z80 carry flag
H       - The Z80 half-carry flag
N       - The Z80 add/subtract flag
P       - The Z80 parity/overflow flag
S       - The Z80 sign flag
V       - The Z80 parity/overflow flag
Z       - The Z80 zero flag
```

For a complete description of the Z80 registers and flags, please refer to the Zilog "Z80-CPU Tecnhical Manual".

The optional "!" is used to access the "old" value of a machine register or flag - the value of the register/flag before the last instruction traced. As ZBUG does not save the "old" values of the pseudo registers (RD, WR, RR, or DR), the "!" wiIl have no affect if used to to refer to them.

The optional "'" used to refer to a register generally means to consider the Z80 auxiliary register of the same name. This refers to the A, F, B, C, D, E, H, L, M, AF, BC, DE, and HL machine registers, and all of the flags. In the case of the other machine registers, the "'" has no affect.

The "'" does have significance with the three ZBUG pseudo registers, and each are described below.

The ZBUG pseudo registers RD and 'RD are set during tracing. Each instruction that accesses memory for a read sets these registers as follows: The RD register is set to the address of the lowest byte accessed by the instruction, and the 'RD register is set to the highest. An instruction that does not access memory for a read will not disturb the contents of either register (the access by the program counter to get the instruction is not considered a read access by ZBUG). For example, if the instruction being traced is a "MOV A,M", both register RD and 'RD will be set to the address contained in the register pair H&L. If the instruction is a "RET", the RD register will be set to equal the address contained in the stack pointer, and the 'RD will

be set to that address plus one. If the instruction is an "LDIR", the RD register will be set to the address contained in the register pair H&L, and the 'RD will be set to that address plus the contents of register pair B&C minus one. Finally, if the instruction is an "LHLD 0100", the RD register will be set to 0100, and the 'RD set to 0101.

The ZBUG pseudo registers WR and 'WR act like the RD and 'RD registers, but are set for memory write accesses.

The RD, 'RD, WR, and 'WR registers facilitate the monitoring of memory read and write accesses.

The ZBUG pseudo registers RR, 'RR, DR, and 'DR are registers utilized by the user and ZBUG to facilitate access to memory with "relocatable" addresses. For example, if a user wishes to debug code in a certain sub-set of a program (such as a single module of the program), he sets register RR (or DR) via the X command to the address of the start of the sub-set or module. The 'RR (or 'DR) register is then set to the end of the module. ZBUG will then always display any addresses which lie between these two values as being a relative positive offset from the contents of register RR (or DR). Any addresses lying outside this range will be displayed as absolute. ZBUG signals the difference by following any relocated addresses with a single quote if relative to RR, or a double quote if relative to DR. Absolute address are not flagged in this manner. To any address (or any expression, for that matter) that the user types with a following single or double quote, ZBUG will add the contents of the RR or DR register, resulting in an absolute address.

If the register RR (or DR) is equal to 0 (its default value), ZBUG will not relocate any addresses displayed, and any typed by the user with the "'" signal for relocation will be taken as absolute values (offset plus a base address of 0).

This feature of ZBUG makes debugging a module with a listing showing only relative addresses a simple matter of typing a relocatable address instead of an offset plus a constant. By using the RR pair to refer to the code of a module, and the DR pair for a separate data module, following a listing is easier. For further discussion, refer to Section 5.1.

When ZBUG is first executed, all registers are initialized. The SP, RD, 'RD, WR, and 'WR are set to point to the bottom of ZBUG - the highest memory address that a debugging program may used. The PC is set to 100 hex. The other registers are set to 0.

UZBUG does not provide the RR, DR, RD, or WR register pairs, or the re-examine ("Rn") feature found in ZBUG.

{ examine the A register }

```
* x a <cr>
00      <cr>
*
```

{ examine and modify the register pair H&L }

```
* x hl <cr>
0001    0 <cr>
*
```

{ examine the B register in various radixes }

```
* x <b
00      ;ra <cr>
'@'     ;rd <cr>
  0     ;rr <cr>
   +0   <cr>
*
```

{ examine and modify the carry flag }

```
* x >c <cr>
0       (<a + 1) > 8 <cr>
*
```

{ examine all }

```
* x <cr>
 AF (000F) BC (0001) DE (1253) HL (0000) FLAGS: .....VNC
 AF'(00F0) BC'(0000) DE'(0000) HL'(0000) FLAGS: SZ.H....
 IX (0000) IY (0000) IR (0000) -   INTERRUPTABLE
 RR (0100) RR'(0000) DR (0000) DR'(0000)
 RD (3900) RD'(3900) WR (3900) WR'(3900)
 SP (0000#) -->          00C3    C370    5734    11F3
 PC (0000') --> MOV      A,M
*
```

4.16 Y - Search

Y [<start>] [, <end>]

Search memory from <start> to <end> for string.

Following this command by up to 32 bytes of data cells in the current mode, separated by semicolons (";") and terminated by a <CR>, search memory within the range <start> to <end>, displaying the addresses (in the current address display radix) of each occurrence. If <end> is omitted, assume 0FFFF hex. If <start> is omitted, assume 0.

UZBUG will display the addresses in hexadecimal.

{ search for 1,2,3,4 throughout all of memory }

```
* y <cr>
1; 2; 3; 4 <cr>
```

```
100F
3451
A003
FF45
*
```

{ assuming instruction mode, look for all calls to location 5 between 100 and 1000 }

```
* y 100, 1000
call 5 <cr>
0134
0562
*
```

4.17 Z - Zap CP/M fcb's

Z <string>

Set up CP/M input as if <string> were part of command at CP/M command level:

A> <program> <string> <cr>

Set up CP/M's TFCB, TFCB+16, and TBUFF with <string>, as defined by the Digital Research "CP/M Interface Guide". If <string> is omitted, clear TFCB, TFCB+16, and TBUFF as also defined.

Note that ZBUG is sensitive to blanks in parsing filenames from <string> - any encountered are assumed to be terminators.

UZBUG does not provide this command.

{ set up a filename }

```
* z file.asm <cr>
*
```

{ set up a string }

```
* z 3/24/78 10:15:00 <cr>
*
```

Section 5

Going Beyond the Basics

Although ZBUG can be used after getting to know how to use a few commands, much can be gained by becoming familiar and comfortable with ZBUG's most flexible, and therefore most consequential, feature - the expression.

The majority of this section will be used to discuss the ZBUG expression (5.1). The remainder will be used to dramatize the ZBUG's potential capabilities by giving specific examples of commands utilizing expressions as arguments, along with suggestions and hints to help utilize ZBUG to it fullest.

5.1 The ZBUG Expression

ZBUG expressions follow a relatively small set of recursive rules. To help visualize these rules, the following is the syntax or structure of all expressions in BNF (Backus Naur Form).

```
<exp>    ::=    <sub-exp> [ <c-op> <sub-exp> ]*
<sub-exp> ::=   <term> [ <t-op> <term> ]*
<term>   ::=    <bool> [ <b-op> <bool> ]*
<bool>   ::=    <factor> [ <f-op> <factor> ]*
<factor> ::=    { <con> | <sym> | ( <exp> ) } ['|"] |
                <u-op> <factor>

<con>    ::=    <reg> | <flag> | <num>
<reg>    ::=    < <register-name>
<flag>   ::=    > <flag-name>
<num>    ::=    <string> |
                <number> |
                <instruction>

<c-op>   ::=
<t-op>   ::=    + | - | ! | ^
<b-op>   ::=    * | / | @ | & | < | >
<f-op>   ::=    ?EQ | ?NE | ?LT | ?LE | ?GT | ?GE
<u-op>   ::=    + | - | # | @ | \ | ^ | !
```

The symbols <sym>, <register-name>, <flag-name>, <string>, <number>, and <instruction> will all be defined in the discussion following.

UZBUG provides for a very restricted expression, defined in the following BNF:

```
<exp>    ::=    <factor> [ + | - ] <factor>
<factor> ::=    <con> | <sym> | ( <exp> ) |
                [ @ | \ ] <factor>

<con>    ::=    <reg> | <flag> | <num>
<reg>    ::=    < <register-name>
<flag>   ::=    > <flag-name>
<num>    ::=    <string> | <hex>
```

Note that the + and - operators are define in Section 5.1.2, the @ and \ in Section 5.1.5, and <hex> is a hexadecimal number as defined in 5.1.7.1.

5.1.1 <exp> and the "_" Operator

As defined earlier, an <exp> (expression) is:

<sub-exp> [<c-op> <sub-exp>]*

-or-

a subexpression followed by zero or more occurrences of a "_" followed by a subexpression

The "_" operator is defined as the CONCATENATE operator in ZBUG. It is dyadic, meaning that it requires two arguments. It operates by concatenating its two arguments in the following manner: Argument #1 (the left one) is shifted right by the length of argument #2. Argument #2 (the right one) is masked to its length (always an integer number of bytes from 1 to 4), low order bytes being masked first. The two resulting values are then or'ed together, forming one value.

Normally, the default length of an arbitrary argument is 4 bytes, the largest value ZBUG can manipulate. Certain values have an implied length, however - such as a register (one or two bytes), or the result of the @, \, ^, and ! unary operators described later.

The concatenate operator has the lowest precedence of all the operators - unless overridden by parentheses, any concatenate operations will performed last, from left to right.

Note that this operator is not available in UZBUG.

Examples....

```
<A _ <BC        ... concatenate the current value of
                ... the A register with that of the
                ... B&C register pair
\100 _ @101     ... concatenate the byte at location
                ... 100 with the word at 101
```

5.1.2 <sub-exp> and the +, -, !, and ^ Operators

As defined earlier, a <sub-exp> (subexpression) is:

<term> [<t-op> <term>]*

-or-

a term followed by zero or more occurrences of a +, -, !, or ^ followed by a term

The +, -, !, and ^ operators are defined as the ADD, SUBTRACT, INCLUSIVE OR, and EXCULSIVE OR operators, respectively. They are each dyadic, requiring two arguments. The operations they perform are two's complement

add and subtract, logical inculsive and exclusive or, respectively. Both arguments are considered to be 4 byte values, with no overflow or underflow indication - the sign bit interpretation is left to the user.

The +, -, !, and ^ operators have the 2nd lowest precedence, and are executed from left to right before any concatenate operators (unless overridden by the use of parentheses, of course).

Note that the ! and ^ operators are not available in UZBUG.

Examples....

```
1 + 2           ... add 1 to 2
35 - 28         ... subtract 28 from 35
63 ! 128        ... inclusive or 63 and 128
63 ^ 128        ... exculsive or 63 and 128
```

5.1.3 <term> and the *, /, @, &, <, and > Operators

As previously defined, a <term> is:

<bool> [<b-op> <bool>]*

-or-

a boolean expression followed by zero or more occurrences of a *, /, @, &, <, or > and a boolean expression

The *, /, @, &, <, and > operators are defined as MULTIPLY, DIVIDE, MOD, LOGICAL AND, LEFT SHIFT, and RIGHT SHIFT, respectively. They perform an integer multiply, integer divide, integer modulo, logical and, left logical shift, and right logical shift, respectively. They are dyadic, requiring two arguments, each considered to be 4 byte values, except for the < and > operators, which use only the low order 5 bits of the 2nd argument to determine the number of bits to shift the 1st argument.

The *, /, @, &, <, and > operators have 3rd precedence, being performed from left to right before the +, -, !, or ^ operations (unless overridden by parentheses).

Note that the *, /, @, &, <, and > operators are not available in UZBUG.

Examples....

```
4 * 2           ... multiply 4 and 2
4 / 2           ... divide 4 by 2
4 @ 3           ... modulo 4 by 3
1 & 5           ... and 1 and 5
5 < 1           ... shift 5 by 1 bit
6 > 1           ... shift 6 by 1 bit
```

5.1.4 <bool> and the ?EQ, ?NE, ?LT, ?LE, ?GT, and ?GE Operators

Earlier, the <bool> was defined as:

<factor> [<f-op> <factor>]*

-or-

a factor followed by zero or more occurrences of ?EQ, ?NE, ?LT, ?LE, ?GT, or ?GE followed by a factor

The ?EQ, ?NE, ?LT, ?LE, ?GT, and ?GE operators are the EQUAL, NOT EQUAL, LESS THAN, LESS THAN OR EQUAL, GREATER THEN, and GREATER THAN OR EQUAL operators, respectively. They are dyadic, requiring two arguments, each considered to be a 4 byte value. Each performs an arithmetic compare of the two arguments (signs are significant), and return a zero if the condition is false and a -1 if true.

These operators have 4th precedence - the highest of the dyadic operators, meaning that unless overridden by parentheses, these operations will be the first dyadic ones performed.

Note that the ?EQ, ?NE, ?LT, ?LE, ?GT, and ?GE operators are not available in UZBUG.

Examples...

```
<A ?EQ 1        ... does register A equal 1 ?
2 ?LT 3         ... is 2 less than 3 ?
-3 ?GE 78       ... is -3 greater than or equal to 78 ?
```

5.1.5 <factor> and the +, -, #, @, \, ^, and ! Operators

Previously, the <factor> was described as:

{ <con> | <sym> | (<exp>) } [' | "] |
<u-op> <factor>

-or-

a constant or a symbol or an expression in parentheses (optionally followed by a ' or "), or a +, -, #, @, \, ^, or ! followed by a factor

This is a recursive definition - describing a factor in terms of an expression or a factor. Later we will discuss constants and symbols and the optional "'", but now to the unary operators, etc.

The +, -, #, @, \, ^, and ! operators are defined as PLUS, MINUS, NOT, WORD INDIRECT, BYTE INDIRECT, EXPLICIT LENGTH or RADIX CHANGE, and INSTRUCTION INDIRECT, respectively. These operators are monadic, requiring only

one argument.

The +, -, and # operators consider their arguments to be 4 bytes values, performing a arithmetic plus, arithmetic minus (two's complement), and a logical not (one's complement), respectively.

The @, \, and ! operators use only the low order 2 bytes of their argument, treating the 16-bit value as an address and returning the word, byte, and instruction value pointed to by the address. These three operators also return an implied length of their results (for use by the concatenate operator) - 2 bytes for @, 1 byte for \, and 1-4 bytes for ! depending on the instruction located by the argument.

The ^ operator is further refined by following it with a modifier.

If the ^ is followed by a "1", "2", "3", or "4", it is an explicit length operator, which means that its argument will be considered to be 1, 2, 3, or 4 bytes long for the concatenate operator (the only operator which uses length).

If the ^ is followed by a "B", "D", "H", "O", "R", or "S", it is a radix change operator, which means that its argument will use binary, decimal, hex, octal, relative decimal, or split octal as the default type-in radix instead of that defined by the use of the RT command of ZBUG.

The unary operators have the highest precedence or all the operators (unless overridden by parentheses), and are performed first.

Note that the +, -, #, ^, and ! operators, and the ' and " modifiers are not available in UZBUG.

Examples....

```
+1              ... return a plus one (no real
                ... effect)
-1              ... return a negative 1
#1              ... return the one's complement
                ... or "not" of 1
@100            ... get the word pointed to by
                ... 100 (the word at location 100)
\<HL            ... get the byte pointed to by
                ... register pair H&L (actually
                ... another way of saying <M,
                ... or contents of register M)
!<PC            ... return the instruction
                ... pointed to by the program
                ... counter
^1(<HL + <A)    ... calculate the value of
                ... register pair H&L added to
                ... register A and specify a
                ... length of one byte
^D(123 + 32)    ... calculate 123 decimal plus
                ... 32 decimal
--1             ... return a minus minus 1 (or 1)
```

5.1.6 "Symbols"

Although ZBUG does not recognize symbols per se, several permanent symbols are defined by ZBUG to facilitate instruction operand encoding by the user. These "symbols" are register names commonly used in Z80 assembler language, and are as follows:

A	=	7
B	=	0
C	=	1
D	=	2
E	=	3
H	=	4
L	=	5
M	=	6
SP	=	6
PSW	=	6
X	=	4
Y	=	4

Although these "symbols" may be used as a constant value in any ZBUG expression, their usefulness is probably confined to use in instruction operands typed by the user.

The special symbol "." refers normally to the current location. When using the E command, . contains the address of the currently open cell. When a breakpoint is encountered from the use of the G command, . conntains the address of the break. When tracing via the T command, . contains the adress of the next instruction (identical to the <PC).

5.1.7 "Constants"

ZBUG provides for a wide variety of constants, including numbers (in the conventional sense), strings, register and flag contents, and even instructions. Each is described below:

5.1.7.1 Numbers

Numbers in ZBUG are one or more digits, followed optionally by a radix modifier ("B", "D", "H", "O", "R", or "S" representing binary, decimal, hex, octal, relative decimal, and split octal, respectively). If the modifier is not present, the number is assumed to be in the current default type-in radix (unless overridden by the use of the radix change operator "^"). In certain cases, a radix modifier typed by the user might be interpreted by ZBUG to be a digit. For example, if the current default type-in radix is hex, and the user types "10B" intending the number 10 binary (2 decimal), ZBUG will read it as 10B hex. The same is true for the "D" modifier. In order to overcome this misinterpretation, the use of the radix change operator will suffice - i.e., ^B10. As an added aid, ZBUG provides the "." as an additional decimal radix modifier.

If digits found in a number do not correspond to the radix assumed, an error will result. Valid digits are:

```
Binary: 0,1
Decimal:        0,1,2,3,4,5,6,7,8,9
Hex:    0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
        (number must start with 0-9)
Octal:  0,1,2,3,4,5,6,7
Relative Decimal:       same as decimal
Split Octal:    same as octal (however, the number is
        checked to conform to 3 digits per byte,
        with the 3rd being 0,1,2, or 3)
```

Note that only hexadecimal numbers are acceptable to UZBUG.

Examples - assuming default type-in radix of hex....

```
10              ... 10 hex
10.             ... 0A hex (10 decimal)
345S            ... 0E5 hex (345 split octal)
345O            ... 0E5 hex (345 octal)
```

Improper Numbers....

```
;D12A           ... "A" is not a decimal digit
3456S           ... the "4" is not proper in the
                ... 3rd place of a group of 3
                ... (i.e., 456 is not 1 byte value)
A01             ... does not start with 0-9
```

5.1.7.2 Strings

ZBUG provides the capability of using ASCII character strings as constants. These are a string of characters (not including a <CR>) bracketed by a pair of matched single or double quotes. Strings may be of arbitrary length, but the last four characters are the only ones used.

Examples....

```
"this is a string"      ... only got "ring"
'A'                     ... got a 41 hex value
'AB'                    ... and a 4142 hex
```

Improper Strings....

```
"oops'          ... quotes are not matched
"bad one        ... missing quote
```

5.1.7.3 Registers and Flags

Since the Z80 machine registers and flags, and the ZBUG pseudo registers contain values, it stands to reason that the user should be able to access their values in expressions easily. ZBUG provides this useful capability.

The general form to access a register value is:

< [!] ['] <register-name>

where <register-name> is as described in Section 4.15

The general form to access a flag value is:

> [!] ['] <flag-name>

where <flag-name> is as described in Section 4.15

The optional "!" specifies to ZBUG to return the value of the register or flag before the last instruction trace. ZBUG saves all Z80 machine registers and flags before tracing an instruction to provide access to the previous and current values. This facilitates checking to see if a register/flag value changed during the last instruction traced. For example, the expression ">!C ?NE >C" would return a -1 if the carry flag had changed. The "!" option has no effect if used to access a ZBUG pseudo register, as these registers are not saved during tracing.

The "'" option specifies the auxiliary register set of the Z80, as described in Section 4.15.

The register values have an implicit length associated with them - one or two bytes, depending on the register specified.

Examples....

```
<A          ... return the byte value of
            ... register A
<AF         ... return the word value of
            ... register pair AF (also
            ... known as the PSW)
>Z          ... return the bit value of
            ... the zero flag
```

5.1.7.4 Instructions

As an instruction certainly has a numeric value, ZBUG provides the means to use an arbitrary instruction as a numeric constant.

The form is a "[" followed by the instruction and any operands (separated from the mnemonic by a space (" ") or tab (control-I)) and terminated by a "]". If the "]" is not immediately following the instruction, or if the instruction is not properly formed, an error will result. See the XITAN Macro Assembler User's Manual for the correct formats of instructions.

Note that instructions as numeric values are not available in UZBUG.

Examples....

```
[MOV A,B]        ... return a 78 hex
[CALL 5]         ... return a 0005CD hex
[MVI B,1]        ... return a 0106 hex
```

Improper Instructions....

```
[MOV A,B)        ... need that "]" !
[INR Z]          ... what's a Z ?
```

5.2 Advanced Ideas

Learning to use ZBUG should be a growing process. After learning and using a few commands, such as those demonstrated in the example of Section 3.3, more advanced features of ZBUG can be incorporated in the user's repetoire of well-used capabilities.

The following is an informal discussion, intended to give the user an idea of how to get the most out of ZBUG, After becoming familiar with ZBUG's sophistication, less time will be spent "fighting with the debugger than the debuggee". Please note that these suggestions are no more than quidelines....

When first entering ZBUG, use the "M" (mode) and/or "R" (radix) command to set up the mode/radix environment you're most comfortable in or will be working in most. For debugging a program with a listing, instruction mode and hex radix are usual. Then, if it becomes convienient later to look at some value in a different mode/radix, the temporary settings available with the "E" (examine) and "X" (examine register) commands will usually be sufficient.
With the "E" command, it is possible to change the mode and radix together.

{ this is a demonstration of the mode/radix changing within the "E" command }

```
* e 100 <cr>
0100:   01      ;mi rd <cr>
0100:   LXI     B,    0 ;m4 rb <cr>
0100:   00101111000000000000000000000001        ;ro <cr>
0100:   05700000001     ;m2 <cr>
0100:   000001  ;rr <cr>
0100:       +1  <cr>
*
```

{ radix changing within the "X" command }

```
* x hl <cr>
1234    ;ra <cr>
'^P4'   ;rs <cr>
022:064 ;rb <cr>
0001001000110100        <cr>
*
```

The more comfortable you are with expressions, and what they're capable of, the less you'll have to type to get the job done, and the more you'll be able to accomplish. For example, remembering that "<BC" means "the value in register pair B&C", whenever you want to see the value pointed to by the address in B&C, you can use "<BC" as the address for an application of the "E" command. This way you save yourself the trouble of examining the contents of the B&C register, and then typing that value. This of course applies for any address, data, count, etc. value expected by any command,

for any register or flag value, and for instructions and byte or word memory values. In a sense then, these special values may be thought of - and used instead of - numbers.

```
    { examine the instruction pointed to by the program
counter }

    * e <pc <cr>
    0105:   ANI     0F      <cr>

    { using the "L" command, list the ASCII string, whose
starting address is in the word pointed to by register H&L,
and whose lenth is in the byte following the address word }

    * l @ <hl, \ (<hl + 1) <cr>
    4DA2:   100.PRINT X
    *
```

The "S" command expects a boolean expression in order to set a trap or conditional display. This may actually be ANY expression - ZBUG evaluates it and traps or displays if the result is non-zero. Due to the flexibility of the expressions recognized, the trap/conditional display capability is extremely powerful and versatile.

```
    { set a trap to fire when the next instruction to be
executed is a PCHL and the address in register H&L is 0 }

    * s0,(!. ?eq [pchl]) & (<hl ?eq 0) <cr>
    *

    { set a trap to fire if the last instruction wrote in
the area of 0 to 0FF - note that we'll only use the low byte
register of the write access pair, and we can't be certain
that an LDIR or LDDR instruction wrote here due to those
instructions' "wildness" }

    * s0, (<wr ?ge 0) & (<wr ?le 0FF) <cr>
    *

    { set a trap to fire whenever the carry flag is set (not
0) }

    * s0, >c <cr>
    *

    { set a trap to fire whenever the zero flag changes }

    * s0, >!z ?ne >z <cr>
    *
```

If you're going to be debugging a single module of a large program, for which you've got a listing with only relative addresses, the relocation registers can save you a lot of time and trouble. All that is necessary is to know the absolute address of the start of the module (from a linkage editor load map), and set the RR register (or the DR) to that address via the "X" command. If the ending

address (or start address plus length) of the module is known, set the 'RR (or 'DR) to it. From then on, if ZBUG displays an address that is in the module, it will be displayed as a relative offset from the start, followed by a ' (or "). To access an address within the module, you only have to enter the relative offset, followed by a ' (or "). This alleviates a common frustration of trying to figure out the absolute address. In addition, any addresses outside the module will be displayed as absolute.

Since it is possible to have more than one relocation base (refer to XITAN "Z80 Relocating/Linking Assembler User's Manual") for a module, such as a code base and a separate data base, the RR-'RR pair can be used for one (code ?) and the DR-'DR pair for another (data ?).

{ knowing the absolute address, examine a routine to see it with absolute addresses }

```
* e 1fed <cr>
1FED:   PUSH    PSW     <lf>
1FEE:   LXI     B,1     <lf>
1FF1:   RAR             <lf>
1FF2:   JRNC    1FF8    <lf>
1FF4:   LBCD    0455    <lf>
1FF8:   LXI     H,0     <lf>
1FFB:   RAR             <lf>
1FFC:   JRNC    2001    <lf>
1FFE:   LHLD    0459    <lf>
2001:   POP     PSW     <lf>
2002:   RET             <cr>
*
```

{ okay, now set up RR and 'RR }

```
* xrr <cr>
0000    1fed <cr>
* x'rr <cr>
0000    2002+1 <cr>
*
```

{ now look at the routine, remembering that addresses refered to inside the routine will be relative offsets to register RR, and those refered to outside will be absolute }

```
* e 0' <cr>
0000':  PUSH    PSW     <lf>
0001':  LXI     B,1     <lf>
0004':  RAR             <lf>
0005':  JRNC    000B'   <lf>
0007':  LBCD    0455    <lf>
000B':  LXI     H,0     <lf>
000E':  RAR             <lf>
000F':  JRNC    0014'   <lf>
0011':  LHLD    0459    <lf>
0014':  POP     PSW     <lf>
0015':  RET             <cr>
*
```

Remember the current type-in radix that you're operating under - it is easy sometimes to enter a number thinking it is one value, while ZBUG calculates another. For example, the sequence "10B" alway means 10B hex if the current type-in radix is hex, and not 10 binary with a "B" modifier. The same goes for "10D" - it's not 10 decimal.

To alleviate this confusion, use the radix-change operator.... ^B10 for 10 binary, ^D10 (or 10.) for 10 decimal, etc.. Since the radix-change operator is a unary operator, you can follow it with an expression in parentheses to include more than one number.

ZBUG uses this, in fact, to enclose each trap and conditional display expression entered. This insures the user that the expressions will be evaluated using the type-in radix in effect when the expression was originally entered, in case the user decides to change the defualt later.

Although the use of carriage return and line feed are fairly standard for examining memory (such as Digital Equipment Corporation's DDT), additional flexibility may be gained with the escape ("call") and form feed ("return") methods of closing a currently open cell and moving on. They are especially useful when examining a section of code with a jump or call instruction - facilitating examining the code at the jump or call address with one keystroke, and returning to the instruction following the jump or call with another.

Appendix A

A Quick Reference to the Commands

The following is a quick reference to the ZBUG command set and special characters.

```
C <expression>                  - "Calculate"
D <address>,<count>             - "Display"
E <address>                     - "Examine"
F <address>,<count>,<value>     - "Fill"
G <address> [,<break>]*         - "Goto"
I <instruction>                 - "Execute"
L <address>,<count>             - "List"
M <mode>                        - "Mode"
O <name>,<bias>,<rel>           - "Open"
P <address>                     - "Put string"
Q                               - "Quit"
R <type><radix>                 - "Set radix"
S <id>,<expression>             - "Set trap"
S* <id>                         - "Clear trap"
SD <id>,<expression>            - "Set cond. disp."
SD* <id>                        - "Clear cond. disp."
SW <time>                       - "Set wait"
T <address>,<count>,<op>        - "Trace"
X <reg/flag>                    - "Examine reg/flag"
Y <address>,<address>           - "Search"
Z <string>                      - "Zap CP/M fcb's"
```

```
control-D       - "end ASCII string"
control-E       - "halt trace"
control-T       - "show current trace location"
<CR>            - command terminator, close cell
                - and close reg/flag
<LF>            - open sequential cell
<BS>            - open last sequential cell
<ESC>           - open "called" cell
<FF>            - open "returned" cell
;               - command separator, re-open current cell
```

Note that the "I", "R", "S", "S*", "SD", "SD*", and "Z" commands, and the control-T feature are not available in UZBUG.

Appendix B

Error Messages

The following is a list of various ZBUG error messages.

All errors encountered are flagged by ZBUG with "*** ERROR :" followed by a two digit error code. An explaination of each error code is follows:

00: "Unknown Command Name" - An unidentified command character was encountered

02: "Can't use byte addr" - While in E command and currently in byte mode, an <ESC> was used to close the open cell. ZBUG will not permit this operation.

03: "Bad mode" - While in E command and using ';' to close the cell, an inproperly formed 'Mn' was found.

04: "Missing arg" - ZBUG expected an argument for the command that was omitted.

05: "Bad cmnd mod" - An improper modifier for the command was encountered.

06: "Missing ',', ';', or <CR>" - ZBUG expected to find a separator or command terminator.

07: "Expr too long" - The specified expression for the S or SD command exceeds 59 characters.

09: "Bad reg" - Unknown register name in X command.

10: "Bad radix" - While in E or X commands using ';Rn' to close, an improperly formed 'Rn' was found.

11: "Bad flag" - Unknown flag name in X command.

12: "Bad mnemonic delimiter" - An improper character was found delimiting an instruction mnemonic.

13: "Bad mnemonic" - An unknown instruction mnemonic was found.

14: "Bad 'hlt'" - The sequence "mov m,m" and related indexed instructions are not premitted.

15: "Can't use 2 index regs" - An instruction may not use more than one index register.

16: "Bad rel addr" - The relative branch is out of reach.

17: "Bad reg / disp (index)" - The register or displacement(index) operand is improperly formed.

18: "Too many args" - Too many arguments were encountered.

19: "Bad file name/ext" - The file name/extension specified in the O command is improperly formed.

20: "Can't open file" - The file specified in the O command doesn't exist.

21: "Read error" - A read error has occurred while reading the file specified in the O command.

22: "Record error" - An improperly formed .HEX or .REL record was found while loading the file specified in the O command. Try re-assembling the program.

23: "Program too large" - The program being loaded with the O command has attempted to load within ZBUG's bounds.

24: "Improper Relational Operator" - ZBUG found a relational operator ('?XX') that was improperly formed.

25: "Missing ')'" - ZBUG found a '(' that was not matched by a ')'.

26: "Improper or Missing Unary Operator" - ZBUG found an improperly formed radix change/explicit length operator, or expected a unary operator and didn't find it.

27: "Improper Register Name" - ZBUG found a register reference with an unknown register name.

28: "Improper Flag Name" - ZBUG found a flag reference with an unknown flag name.

29: "Missing String Terminator" - ZBUG could not find a matching 'string terminator before finding a <CR>.

30: "Improperly Formed Number" - ZBUG found an improper digit for the assumed radix.

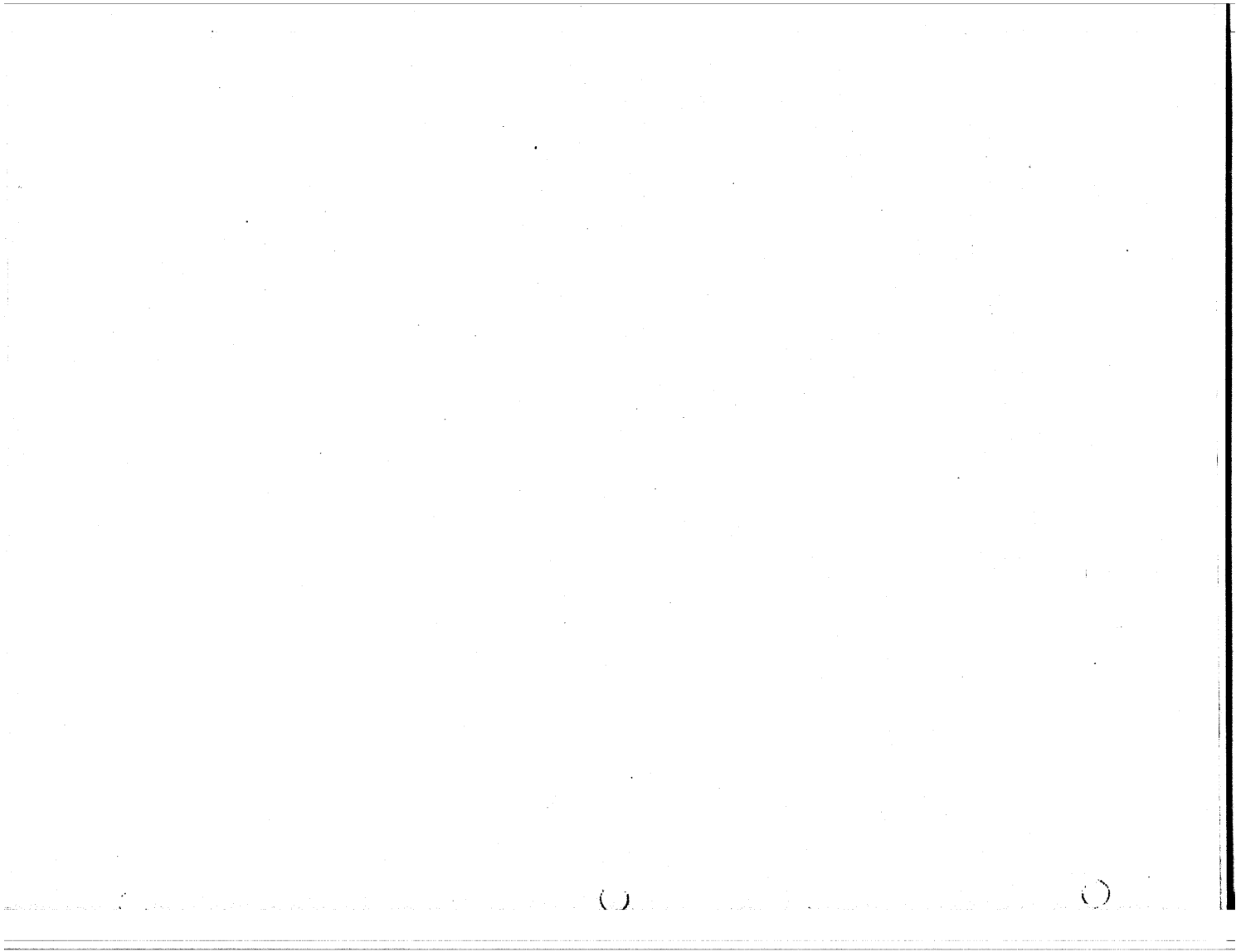
31: "Unidentified Symbol" - General catch all for unrecognizable constructions.

32: "Missing ']'" - ZBUG couldn't find a ']' while processing an instruction "number".

98: "Operand Stack Underflow" - ZBUG system error. If this error happens repeatedly, please gather any

pertinent information (command executing, user entered information, ZBUG generated results, etc. resulting in the error) and notify Xitan, Inc., attention Manager of Tecnhical Services.

99: "Operand Stack Overflow" - The expression ZBUG is evaluating is too complicated - parentheses are nested too deeply or too many operations are pending.



TDL Z80 LINKER

User's Manual

(Manual Revision 0)
March 24, 1978

Written by David S. Hirschman

Table of Contents

1 Introduction

LINKER is a TDL utility program that can bind together individually compiled modules of a program into a single file that may be loaded and executed by the CP/M * operating system.

There are many advantages to the practice of linking together separately compiled modules instead of working with a single, large program. A large program may be decomposed into small modules which may be edited and compiled more quickly. For example, to correct a bug, the programmer need only re-compile the affected modules and re-link the program, instead of re-compiling the entire program. Generally, the linking process is faster than compilation.

It often happens that a routine is used in several programs, a special I/O routine or COSINE function, for example. Instead of copying the source code for this routine into each program, it may be compiled once and then linked in wherever it may be required. Furthermore, using LINKER, routines written in different languages may be combined into a single program.

The Z80 Macro Assembler and FORTRAN IV can produce "libraries", or files containing more than one separately compiled module. LINKER offers methods for including all or only some of the modules in a library into the program.

The remainder of this guide describes how to use LINKER. An overview of LINKER concepts and operation is offered in section 2. The input format to LINKER is defined in section 3.

* CP/M as it appears in this manual is a Registered Trademark of Digital Research.

2 Overview of LINKER Operation

LINKER takes as input, FILEs which contain one or more separately compiled MODULEs. Files containing many modules concatenated together are referred to as LIBRARYs.

Each module has a name. In Z80 Assembler, the .IDENT pseudo operation is used to declare the module name. Its use is highly recommended, as the default module name is ".MAIN.", and duplicate module names in a program are not allowed. The other translators assign a module name automatically.

Each module is made up of SEGMENTs (also called "relocation bases"). Segments are the basic units of code and/or data involved in the linking process. After LINKER is aware of what modules are to be included in the program, it assigns an absolute memory address to each segment in each module. Any code in each segment is relocated so that it will execute at the address to which it is assigned.

Several kinds of segments may be contained in a module. The main code segment, usually containing all of the executable code in the module, has the same name as the module itself. The main data segment of each module also has the same name as the module, preceded by a quote ('). For example, a module named ARCTAN would contain a code segment named "ARCTAN" and a data segment named "'ARCTAN".

All of the other segments in each module are common areas, usually containing only data, which may be shared by other modules. One of these segments is named ".BLNK.", and is referred to as the "unlabeled common". This is the common block that will be created by FORTRAN when the programmer doesn't supply a specific name for a common block. All of the other common blocks have names specified by the programmer.

One of the major features of the LINKING process is that each separately compiled module may access code and data defined in other modules. An INTERNAL symbol is one whose address is available to modules other than the one in which it is defined. Symbols which are not INTERNAL are invisible to other modules. An EXTERNAL symbol is one which is used in a module, but is actually an INTERNAL symbol in another module. All EXTERNAL references must be satisfied by INTERNAL declarations in another module, with two exceptions: symbols may be explicitly defined using the /DEFINE option (section 3.7), and some symbols are pre-defined by LINKER (see Appendix B)

An ENTRY point is an INTERNAL symbol which comes into play in library search mode. In this mode of operation only those library modules having ENTRY points which are referenced as EXTERNAL symbols by one or more already linked modules are included in the program (see section 3.6).

3 LINKER Input Format

MODULE, SEGMENT, and SYMBOL identifiers

An identifier is a string of characters from the following set:

A-Z, 0-9, #$%*&'?+-\^~{|}!.:<>[]_

Normally, an identifier consists of no more than six characters. However, an identifier for a .DATA. segment of a module (as discussed in the previous section) is preceded by a quote (').

Identifiers may not contain blanks. Lower case letters, when used, are automatically translated into upper case. The first character of an identifier may not be a number 0 - 9. The following are examples of valid identifiers:

```
PROG5A
SORT-3
'SORT-3     (a .DATA. segment name)
FOO$$$
```

The following are not valid identifiers:

```
34ABC     - an identifier may not begin with a number
CHECKERS  - too many characters
NIM A     - contains a space
```

FILE NAME

A file name has the following format (with brackets [] indicating optional portions):

[device:]name[.extension]

The "device:" indicates on what disk drive the file resides. If present, it must be one of "A" through "P". If omitted, the logged-in disk is assumed. If LINKER can't locate an input file on the specified disk, it will try drive A.

The file "name" is required, and must consist of no more than eight characters from the character set given above for identifiers, except that the characters <>.:[]_ may not be used.

The ".extension" indicates what the type of the file is. It may consist of no more than 3 characters, from the same set of characters allowable in the file "name". The defaults for ".extension" are defined on page 7.

16 BIT VALUE

A 16 bit value may be expressed as a literal or as a number. A literal is one or two characters enclosed in quotes, for example: "V1".

A number may be expressed in several different bases, as shown in the table below. A radix character immediately following the number indicates which number system is being used:

Base	Radix	Valid Digits	Valid Range
hex	H	0-9 , A-F	0 - 0FFFF
decimal	.	0-9	0 - 65535
octal	O	0-7	0 - 177777
binary	B	0 and 1	16 digits

If the trailing radix character is omitted, "H" (hex) is assumed. All numbers must begin with a numeric digit (0-9). A preceding minus sign indicates a negative number. In this case, a two's complement representation is used.

The following are examples of 16 bit values:

```
14170     - an octal number
0C1B5     - a hex number
-55.      - a negative decimal number
"A"       - a one character literal
```

The following are not valid 16 bit values:

```
100000.   - decimal number too large
960       - invalid octal digit
"AB       - missing closing quote
C1C2      - does not begin with a digit
```

INITIATING LINKER

LINKER may be used interactively, or input may be given as it is executed:

LINKER <commands> <cr>

This means that LINKER may used in a SUBMIT file.

To use LINKER in the interactive mode, simply enter

LINKER <cr>

on the console. LINKER will read commands from the console, prompting with an asterisk "*". All input is stored uninspected until a carriage return is typed. The standard line editing features of CP/M * (rubout, CTL-U, CTL-C, CTL-E, etc.) are available. If a CTL-T is found on any line, the entire command being entered is aborted.

A disk file containing all or only part of a command may be inserted into the input at any point by preceding the disk file name with an "@". The default file extension is ".LNK". These disk files may not contain further "@" specifications. The most common use of this feature is to prepare a file containing a complete command; then,

LINKER @<file name> <cr>

links the program. Usually, these ".LNK" files may be prepared once for a given program and used over and over again, greatly simplifying the whole process.

All LINKER commands have the same format, regardless of whether the interactive mode is used. Commands are separated by a semi-colon ";". LINKER terminates when it receives the "Q" command (quit). For example,

<command> ; <command> ; <command> ; Q

LINKER also terminates when input provided with its execution is exhausted.

If an error is found, the current input line is echoed with two question marks inserted after the point at which the error was detected. This is followed by an error message (see Appendix A). The command must then be re-entered.

All input is free format. Blank lines are ignored, and a command may extend to any number of lines. All lower case letters are automatically translated to upper case. Comments may be included with input from any source by using an asterisk "*". When an asterisk is encountered, all remaining characters on the same line are ignored.

If a CTL-C is typed while LINKER is running, it will quit and return to the monitor. If CTL-E is typed, the current command is aborted, and LINKER will prompt for more input if it is being used interactively.

3.1 Command Syntax

Each command to LINKER links one program, and is of the format:

```
[<output file> =]
    <input file 1>, <input file 2>, ... , <input file m>
    /<option 1> /<option 2> ... /<option n>
```

LINKER links together appropriate modules from the input files to create the output file, under control of any options present. If the program is linked successfully, its name is printed on the console, along with the address of the highest byte used in the program and the program size rounded up to the nearest K (1K = 1024 bytes).

3.2 Output File

The output file is the file which will contain the linked program. The file extension indicates what kind of file is to be produced. If given, it must be one of the following:

COM - Absolute binary core-image file, ready to be loaded and executed by the operating system.

HEX - INTEL "hex" format file (see Appendix I of the Z80 Relocating Macro Assembler User's Manual).

If the .<extension> is not given, ".COM" is assumed. The output file replaces any existing file of the same name.

Examples:

B:PROG1	- A .COM file for PROG1 is placed on disk B.
PROG2.HEX	- An INTEL "hex" file for PROG2 is placed on the currently logged-in disk.

The output file and equal sign following it may be omitted; then, the name of the first input file is used, and an extension of .COM is assumed.

3.3 Input Files

Each <input file> may contain either a single compiled module, or may be a library containing many compiled modules. Normally, all modules contained in each <input file> will be included in the output file, but this default action may be overridden as explained below. The <input file>s must contain all modules that are to be included in the output file, unless the /SEARCH option is used (see section 3.6).

If the file extension is not given, ".REL" is assumed. Of course, all files must contain only compiled, relocatable object modules, in either ascii or binary format.

A module selection clause may optionally be added immediately after each input file name, to indicate that only some of the modules within the file are to be linked. It has two possible formats:

(INCLUDE <module 1>, <module 2>, ... <module n>)

which causes only the named <module>s to be included in the output file, and

(EXCLUDE <module 1>, <module 2>, ... <module n>)

which causes all modules in the library EXCEPT the listed ones to be included in the output file.

3.4 /MAIN Option

This option specifies the main module of the program. Its format is:

/MAIN <module name>

The main module must have a defined starting address. This is done in Z80 Assembler by supplying a label with the ".END" pseudo op. The other translators automatically supply a starting address. Execution of the program will begin at this address.

If the /MAIN option is omitted, LINKER looks for a global symbol named .MAIN. and uses this for the starting address if found. If not, the first module encountered in the input files which has a defined starting address is assumed to be the main module of the program.

3.5 /MAP Option

The /MAP Option may be used to obtain a printout of the memory map on the list device. Reports can be selected that show the memory addresses assigned by LINKER to the segments and symbols in the linked program, or that describe the modules that were linked.

The format of the /MAP option is:

/MAP <flag 1> <flag 2> ... <flag n>

The <flag>s control what items will be included in the memory map report, as follows:

G - Global symbols (i.e. all internal symbols of all loaded modules). The symbols are listed in alphabetical order, with their assigned addresses. The address shown is the address that will be used for all references to this symbol. This may not be the same as the address where the symbol is loaded, if the /ACTUAL option is used.

S - Segments. All of the program segments are listed in alphabetical order, and the assigned address and size is given for each. If the segment is to be relocated so that it will execute at an address different from its assigned one, via the /ACTUAL option, this address is given also.

A - All. This option combines the information given by the S and G flags. All segments are listed, in order of ascending memory address. Each segment is followed by all of the global symbols contained within that segment, again listed by ascending memory address. Absolute symbols are listed under a dummy segment named .GLOB.

M - Modules. Each module is listed, along with its ID number, version and revision number, and date and time assembled (older versions of the Z80 Assembler do not output the information needed to generate this report. The .PROGID pseudo op is used to create this information for each module).

If no <flag>s are given, /MAP A is assumed.

3.6 /SEARCH Option

This option causes library files to be searched in order to satisfy external references which remain unresolved after all modules contained in the input files have been linked. The format of the option is:

/SEARCH <library 1>, <library 2>, ... <library n>

Each <library> has the same syntax as the input files of section 3.3. INCLUDE and EXCLUDE clauses may be used.

A module in a library is loaded when one or more of its ENTRY points (see section 2) are referred to by other modules, but have not yet been defined anywhere. As long as undefined symbols exist, all specified libraries are searched iteratively in the order given, until a complete pass over the libraries yields no new modules to be loaded. That is, if loading a library module creates new unresolved symbols, all of the libraries may be searched again in an attempt to find it.

When FORTRAN IV modules are included in a program, the FORTRAN library "LIBRARYS.REL" is automatically added to the end of the list of libraries to be searched. It must be present on the logged-in disk or on drive A. This library is designed to be searched in a single pass, and error #31 (see Appendix A) may result if an additional pass must be made over it. Therefore, it may necessary to design any other libraries that are to be searched so that only a single pass is required to pick up all needed modules.

3.7 /DEFINE Option

This option may be used to give values to symbols which are not defined by any module in the program. These defined symbols are then used to resolve EXTERNAL references made by the program modules.

The syntax of this option is:

```
/DEFINE <symbol 1> = <value 1>,
        <symbol 2> = <value 2>,
                   .
                   .
        <symbol n> = <value n>
```

Each symbol is given a 16 bit value. This value could represent a constant, or an absolute address.

The format is:

```
/ACTUAL <segment-1> = <address-1>,
        <segment-2> = <address-2>,
                    .
                    .
                    .
        <segment-n> = <address-n>
```

Each segment, which will be loaded wherever it would normally be loaded, will be relocated to execute at the given address. All references from other segments into them will also be relocated.

APPENDIX A - LINKER Error Messages

A few LINKER error conditions are indicated by a short message which should be self-explanatory. For the rest, an error number is given which may be looked up in the table below. In the case of a syntax error, the input line containing the error is echoed, with two question marks "??" following the point where the error was detected. Other errors may be flagged as occurring in PASS 1 or PASS 2.

Many of the error messages involve a problem with a disk file. In this case, the name of the disk file is given, as well as a byte offset (in hex) indicating the position in the file where the error was detected.

Any error codes not appearing in this table are diagnostic errors indicating a bug in LINKER. Try running LINKER again. If the error persists, please collect the relevant information (error message, LINKER version date, input files, etc.) and notify the Technical Assistance Manager at Technical Design Labs.

Error Codes

1 - Expecting equal sign.

2 - Expecting "/" or ";". The command parser has reached the end of the input files, and is trying to read the options.

3 - Bad option name. See sections 3.4 and following.

4 - Option not implemented. The version of LINKER you are using does not contain this option yet.

5 - Expecting identifier. See Section 3 for an explanation of correct identifier format.

7 - Wrong digits in number. Which digits are valid depends, of course, on the radix you are using. See Section 3.

8 - Number or literal too large. All numbers and literals must be able to fit into 16 bits. See section 3.

9 - Token too large. The string of characters you entered at this point is too long to possibly be any kind of valid input.

10 - Expecting "device:" or "file" name. A proper file name should appear in the input at this point (see section 3, file name format).

11 - Invalid "device:" specifier. Valid device specifiers are "A:" through "P:".

12 - Invalid file name. A file name must consist of no more than eight characters from the proper character set (see section 3, file name format).

13 - Invalid file extension. A file extension must consist of no more than three characters. An output file may only have extensions ".HEX" or ".COM".

14 - Expecting 16-bit value. A number or literal must appear in the input at this point.

15 - Incorrect INCLUDE or EXCLUDE format. Either you did not give one of the key words INCLUDE or EXCLUDE, or there is an incorrect module ID, or the closing right parenthesis ")" is missing.

17 - "@" inside @ file. Disk files containing commands and used via an "@" may not contain further "@" specifications.

20 - Insufficient memory. There was not enough free memory available for LINKER to use for its symbol and segment tables. Therefore, the program could not be linked.

31 - Duplicate segment. The indicated segment appears more than once in the input modules. Did you remember to use the .IDENT pseudo op in Z80 Assembler programs? Another way this error can occur is if FORTRAN IV is being used and multiple /SEARCH passes are made over LIBRARYS.REL. See section 3.6.

34 - Undefined segment. A segment which you referred to in the /LOCATE or /ACTUAL options was never encountered in the input files.

40 - Can't close output file. Is the disk write protected?

41 - Error in extending file.

42 - No space for output file. There is not enough space on the disk to hold the output file.

43 - No directory space. The disk upon which the output file is to be placed doesn't have enough room in the disk directory.

45 - Can't open output file. This error may be caused by a full directory, or by a protection failure.

46 - Loading below 100H in .COM file. A .COM file is organized so that the beginning of the file corresponds to memory address 100H, since the operating system always loads a .COM file at this address (see Appendix D). Thus, nothing may be loaded below this address. This error may be caused by a /LOCATE to an address below 100H.

50 - Expecting module record. The input file was supposed to contain a module record at this point, but did not. This error often occurs when there is trash at the end of the previous module in a library file.

51 - Invalid record type. The input file contained an incorrect .REL record type at the indicated offset.

53 - Undefined symbols exist. All of the listed symbols will have to either be made INTERNAL symbols of some module or defined via the /DEFINE option.

54 - Missing starting address. You did not use the /MAIN option, symbol .MAIN. did not exist, and none of the program modules had a defined starting address.

55 - The main module (as given by the /MAIN option) has no defined starting address. Be sure to give a starting address with the .END pseudo op in Z80 Assembler programs.

56 - The main module (as given by the /MAIN option) was never encountered in the input files; therefore, no starting address could be determined.

57 - Can't recognize module. There is garbage in the input file at this point. Are you sure this file is a valid .REL file? If all else fails, try re-compiling.

58 - Can't process FORTRAN. The version of LINKER you are using can't link FORTRAN modules.

60 - Duplicate input file. Each input or library file can appear only once in a command.

64 - FORTRAN symbol number out of range. This and the following two errors usually indicate a smashed FORTRAN .REL file. Try re-compiling.

65 - Bad FORTRAN relocation base type.

66 - Bad FORTRAN op code.

70 - Duplicate symbol. The indicated global symbol is defined in more than one module.

79 - Program won't fit into memory. This program won't fit into the address space of a 16-bit micro-computer. Either it is simply too large, or you created large wasted areas of memory by using the /LOCATE option.

80 - Expecting carriage return. The indicated input file was supposed to have a carriage return at the given location, but did not. Are you sure this is a valid .REL file? Try re-compiling the program if all else fails.

81 - Expecting line-feed in input file.

82 - Expecting ASCII character. The input file did not contain a valid ASCII character where it was supposed to.

83 - Bad Checksum. Z80 Assembler *.REL* files contain checksum bytes after each record which are used to validate the data that is read from them. A checksum error usually indicates a file that is corrupted with errors: try re-compiling.

85 - End of input file. The end of the indicated file was reached unexpectedly.

87 - Empty input file. The indicated input file was totally empty, except perhaps for some filler characters.

Appendix B - Pre-Defined Symbols

There are a few global symbols which are pre-defined by LINKER before the linking process begins. They are listed below. The user should not attempt to define these symbols himself, as a duplicate symbol error (code #70) will result. Future versions of LINKER may have more of these symbols. They will be of the form .XXXX., so the use of symbols of this form should be avoided.

Pre-Defined Symbols

.FREE. - This symbol points to a word which contains the address of the first free byte in memory above the program. It is useful when the programmer wishes to make use of free memory at execution time. When a "/LOCATE .DATA. = <addr>" is done (i.e. data segments are assigned to a separate memory location), .FREE. points to the first free byte above the data area.

If FORTRAN IV modules are included in the program, many other symbols will be defined via modules brought in from LIBRARYS.REL. The reader is referred to the TDL FORTRAN IV User's Manual for details.

Appendix C - Syntax Summary

Below is a brief summary of LINKER input syntax, in a modified BNF format. The symbol "::=" should be read as "is defined to be". Angle brackets "<>" delimit meta-linguistic objects, which are themselves defined in a following line. Square brackets "[]" indicate optional input. Curly braces "{}" indicate input which may be omitted or repeated as many times as desired. A vertical bar "|" indicates a choice - the form preceding or following may be used.

```
<LINKER input>      ::= <command> {;<command>} ;Q

<command>           ::= [<output file> =] <input file>
                                          {,<input file>}
                                          {/<option>}

<output file>       ::= <file name>
                        (the extension, if included,
                         must be .COM or .HEX)

<input file>        ::= <file name> [<module selection>]

<file name>         ::= [<device>:] <name> [.<extension>]

<device>            ::= "A" through "P"

<name>              ::= a string of no more than 8 characters
                        from <fset>, beginning with one of
                        "A" though "Z".

<extension>         ::= a string of no more than 3 characters
                        from <fset>, beginning with one of
                        "A" though "Z".

<module selection>  ::= (INCLUDE <module> {,<module>})
                      | (EXCLUDE <module> {,<module>})

<option>            ::= <main> | <map> | <search> | <define>
                      | <locate> | <actual>

<main>              ::= MAIN <module>

<map>               ::= MAP [A] [S] [G] [M]

<search>            ::= SEARCH <input file>
                               {,<input file>}

<define>            ::= DEFINE <symbol> = <value>
                                {,<symbol> = <value>}

<locate>            ::= LOCATE <segment> = <value>
                               {,<segment> = <value>}

<actual>            ::= ACTUAL <segment> = <value>
                               {,<segment> = <value>}
```

```
<module>        ::= <id>

<symbol>        ::= <id>

<segment>       ::= ['] <id>

<id>            ::= a string of no more than 6 characters
                    from <nset>, which does not begin
                    with a number.

<fset>          ::= A-Z, 0-9, #$%*&'?+-\^~{|}!

<nset>          ::= <fset> and <>.:[]_

<value>         ::= <number> | <literal>

<literal>       ::= "<any one or two characters>"

<number>        ::= [-] 0 - 0FFFF[H]
                        | 0 - 65535.
                        | 0 - 177777O
                        | 0 - 1111111111111111B
```

Appendix F - Using LINKER with Z80 Assembler

This appendix is a list of hints which may be of help in setting up Z80 Assembler modules for use with LINKER.

SYMBOLS

Internal and External symbols are created by using the .INTERN and .EXTERN pseudo operations. .ENTRY is used to create entry-point symbols.

SWITCHES

When assembling a module for use with LINKER, do not use the .PABS or .XLINK switches. Do use the .PREL and .LINK switches (these are defaults). You may use the .PHEX switch to get an ASCII .REL file, but using .PBIN (the default) will result in a savings of disk space.

MODULE NAME

Always use the .IDENT operation to give each module a unique name. If you don't, the module will have name .MAIN. Each module in a program must have a unique name.

STARTING ADDRESS

A label should be supplied with the .END pseudo op to define the starting address of the main module of the program. Then use the /MAIN option of LINKER to indicate the main module. Alternatively, make the starting address .MAIN., and declare .MAIN. as .INTERN (FORTRAN does this automatically).

LIBRARIES

Libraries may be created by using the .PRGEND switch. This results in the creation of a new module starting at that point. PIP may be used to create libraries as well, but use the O (object) switch for binary .REL files, and do not use this switch for ascii .REL files. FORTRAN and Z80 Assembler modules, and binary and ascii modules may be mixed together in a library.

MEMORY MAP

If the M report of the memory map is wanted, use the .PROGID pseudo op to define the program name, version number, and revision number.

COMMON BLOCKS

To make a common block, declare the common block name to be an .EXTERN in each module that must reference it. The common should not be declared .INTERN by any module. Then, use .LOC to define the common. For example,

```
            .EXTERN TABLE
                    .
                    .
                    .
            .LOC    TABLE
A:          .WORD   5
B:          .BLKB   10
C:          .ASCII  "ABCDEFG"
            .RELOC
```

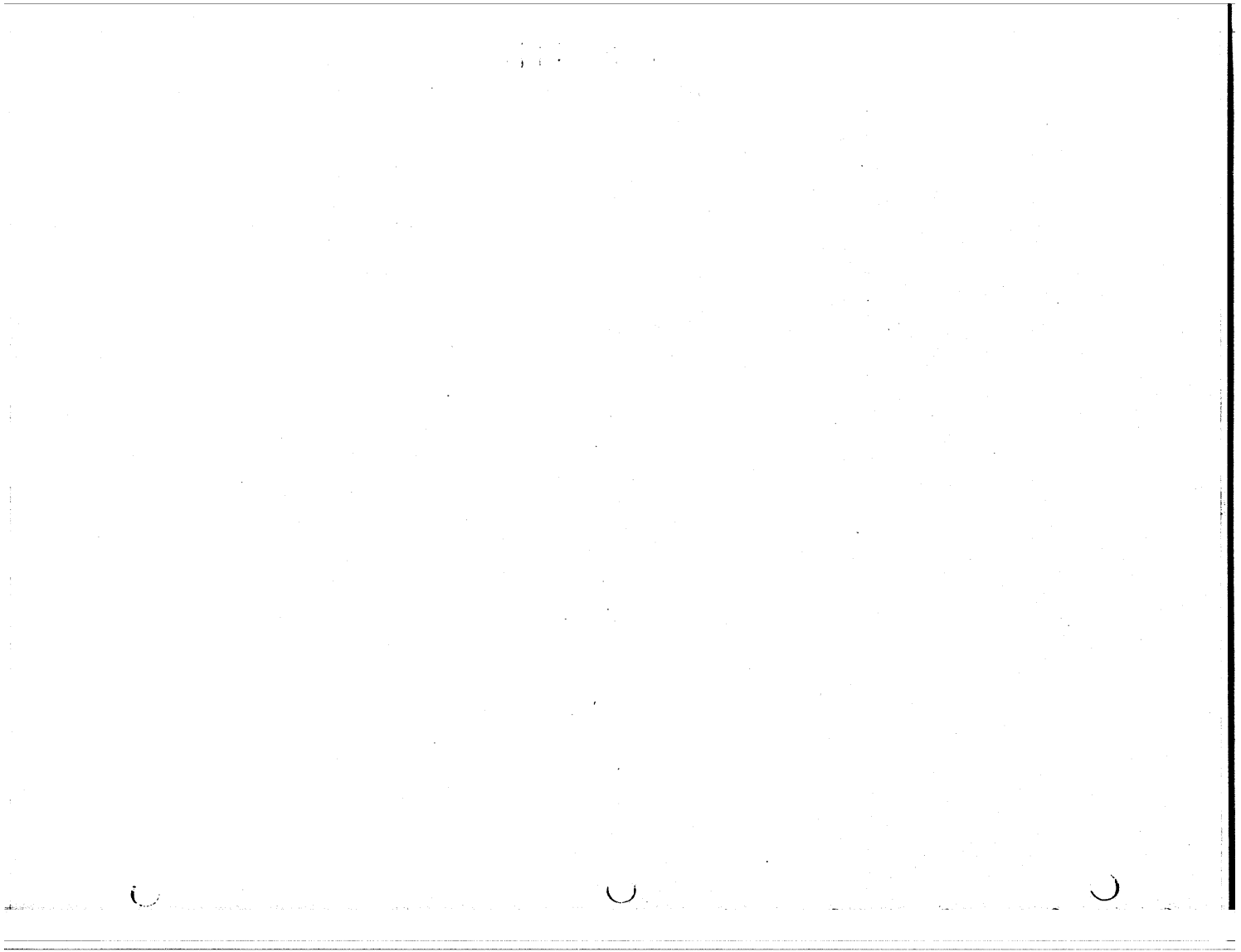
declares a common named TABLE consisting of A, a word, B, 10 bytes long, and C, an ascii string. Remember that FORTRAN will name a common .BLNK. if the programmer does not give it a name.

DATA AREA

Objects are placed into the data segment of a module by preceding them with a .LOC .DATA. The programmer may .LOC .DATA. over and over again in the program: each definition is added on to the end of the previous ones. For example:

```
            .LOC    .DATA.
FOO:        .WORD   0
BAR:        .BYTE   55H
            .RELOC
                    .
                    .
                    .
            .LOC    .DATA.
PTR:        .WORD   TABLE
TABLE:      .BLKW   100
            .RELOC
```

reserves space for four variables in the data segment. LINKER can be instructed to load the data segments and common blocks in a separate area of memory using the /LOCATE option.



READER'S COMMENTS

TDL

TDL Z80 LINKER USER'S MANUAL

In a constant effort to improve the quality and usefulness of its publications, Technical Design Labs, Inc. provides this page for user feedback. Your critical evaluations of this document is our only effective means of determining its serviceability. Please give specific page and line references where applicable.

ERRORS NOTED IN THIS PUBLICATION:

SUGGESTIONS FOR IMPROVING THIS PUBLICATION: (i.e. clarity, organization,convenience,accuracy,legibility.)

MISSING DOCUMENTATION: (i.e. completeness.)

Name----------Date----------

Street----------

City----------State----------Zip Code----------

All comments and suggestions become the property of TDL. Send to Technical Design Labs, Inc.
Dept. of Product Improvements
1101 State Road
Princeton, N. J. 08540

Please indicate in the space below if you wish a reply.