REFERENCE MANUAL

Z8671

SINGLE-CHIP INTERPRETER BASIC/DEBUG SOFTWARE

Preface

This manual describes the Basic/Debug interpreter, a version of Tiny Basic resident in the internal ROM of the Z8671 Single-Chip Interpreter. The first three sections describe Basic/Debug's design considerations, its self-contained editor and language syntax. Sections 4 and 5 give interactive debugging instructions and suggest programming methods that speed up execution time and conserve memory space. The final sections discuss the memory environment and Basic/Debug's interactions with interrupts and external Input/Output drivers.

Because Basic/Debug is a subset of Dartmouth Basic, most of the material covered in this manual will be familiar to Basic programmers. However, Basic/Debug has greater responsibilities than other Basic interpreters. The Z8671 has no other operating system software and therefore depends heavily on Basic/Debug to interact with its environment. Because the Basic/Debug interpreter is stored in the internal ROM of the Z8671, it is defined within the unique hardware characteristics of the microcomputer chip.

This manual introduces the Z8671 hardware environment by describing Basic/Debug's interfaces with machine language code, the memory environment and interrupts. However, to fully utilize the Z8671, more detailed technical knowledge is needed. For example, before a Basic/Debug program can access a machine language subroutine, an assembly language version must be developed, assembled, tested, and stored in the Z8671 system memory. These processes are described in the Z8 PLZ/ASM Language Manual (part number 03-3023-02). Three other helpful documents are The Z8 Microcomputer Preliminary Technical Manual (part number 03-3047-02), the Z8671 Product Specification (part number 00-2180-01), and the application note entitled A Seven-Chip Computer (part number 00-2151-01).

Contents

CONTENTS (cont.)

Section 1
General Information

1.1 Design Considerations

The original Basic developed at Dartmouth College is designed for people who have no previous experience with computers. Because Basic/Debug is a descendant of Dartmouth Basic, it has similar syntax and is easy to learn and use. However, Basic/Debug is designed specifically for process control. Some Dartmouth Basic features which are inappropriate to Z8671 applications have been left out of Basic/Debug. Among these are trigonometric and other transcendental functions, array and character string handling, and fractional numbers. To further conserve memory space, all redundant commands and statement types which can be duplicated by combining other commands have also been eliminated.

However, Basic/Debug allows fast hardware tests, examination and modification of any memory location or input/output port, bit by bit examinations of any port, bit manipulation, and logical operations. The Basic/Debug interpreter can process both decimal and hexadecimal values for input and output. A Basic/Debug program may also access machine language code as either a subroutine or a user-defined function.

Once the application program has been developed and tested, the Z8671 system may be converted from development to automatic mode. When the developed program is stored in a special location in memory, the Basic/Debug interpreter will execute it every time the system is powered up or reset.

1.2 Basic/Debug's Execution Modes

Basic/Debug executes commands in one of two modes: run or immediate. The system is ready to accept a command when the Basic/Debug prompt, a colon, appears at the left edge on a new line at the terminal. To give an instruction in the immediate mode, enter a command keyword, for example, PRINT. The command is executed when the carriage return key is pressed. The command PRINT will leave one line blank on the terminal before the prompt appears on a new line.

Programs are edited and interactively debugged in the immediate mode. Some Basic/Debug commands, such as RUN, LIST, and NEW, are used almost exclusively in the immediate mode. Others, such as GOTO and LET, are used in both modes.

To enter the run mode, enter the command RUN in the immediate mode. If there is a program in memory, it is executed. The system returns to the immediate mode when program execution is complete or interrupted by an error.

1.3 Program Line Syntax

A program is a series of instructions which, when executed sequentially by the computer, accomplishes a specific task. It is entered into memory one line at a time. This section describes the elements of a program line as the computer reads them from left to right. A program line consists of a line number and a command statement, as shown below:

```
100 PRINT "HELLO"
```

The line number indicates that this instruction is part of a program and should not be executed immediately, so Basic/Debug stores the line in memory. Line numbers also indicate the sequence in which the instructions are to be executed. Therefore, if other lines are already stored in memory, Basic/Debug inserts the new line in its numerical place among them. Only values in the range 1 to 32767 are accepted as valid line numbers.

At the terminal device, Basic/Debug separates the line number from the command statement by one space. In memory, however, no space is stored between the line number and the statement. Therefore, if more than one space is entered between the line number and the statement at the terminal, Basic/Debug appears to ignore the extras. If no space is entered, Basic/Debug inserts one before listing the line at the terminal.

Several statements may follow a single line number if they are separated by colons. Packing several commands on one line conserves memory space. The number of commands in the line is not limited, but the line may not contain more than 130 characters.

Basic/Debug ignores the distinction between upper and lower case letters. Therefore PRINT, PrInT and print are all equivalent to Basic/Debug. But in this manual, all example statements are shown in upper case for clarity.

Generally, the command statement has two parts: the command keyword and an argument. In the example line above, PRINT is the command keyword and "HELLO" is the argument. However, Basic/Debug recognizes a wide variety of statements in which either keywords or arguments are omitted, as shown in the following list of valid statements:

```
PRINT
IF C <> USR(A) %500
@%1020 = 100
"THE ANSWER IS";X
```

Basic/Debug recognizes fifteen keywords. Each specifies a statement type which performs one of three actions: assignment to a variable (LET), input or output (INPUT, IN, PRINT), or control flow (IF, GOTO, GOSUB, RETURN, GO@). In the sample program line above, a space separates the keyword PRINT from the argument "HELLO". Although it makes the statement easier to read, the space is unnecessary. Within the statement portion of a program line, Basic/Debug ignores all spaces. Any spaces entered remain in the program and take up memory space, however, Basic/Debug does not recognize that spaces delimit or separate parts of the statement. It looks for other clues which are specific to the command keyword. These delimiters are discussed as each command is defined in Section 3.

The argument portion of a statement may be an expression or, in some cases, another statement. An expression specifies a number or a computation resulting in a number. Elements of expressions are discussed in Section 2. Below are examples of valid expressions:

```
(4096)
A*B*C
@%1020
↑G*100
```

1.4 The Basic/Debug Editor

Basic/Debug supports interactive debugging with a self-contained line editor. It also allows elimination of typing and other errors as a program is entered. Editing is done in the immediate mode. To print a program currently contained in memory, give the command LIST. Then examine the program and make changes and additions using the techniques described below.

Basic/Debug stores program lines in line number sequence. If a line is typed with the same number as a line already in memory, the new line replaces the old one. If only the line number is entered, the line is deleted from memory. Once a line is stored in memory, the only way to change it is to retype the line.

Until the carriage-return key is pressed at the end of the line, the characters entered are temporarily stored in a line buffer. If an error is detected in a line before it is stored in memory, correct it by backspacing through the line buffer to

the mistake and retyping. Backspace by pressing the backspace key or by holding down the control key and pressing H. Each backspace keystroke deletes one character from the line buffer. If more backspaces are entered than there are characters in the line buffer, Basic/Debug deletes the whole line and retypes the prompt on the next line.

If it is necessary to delete a whole line before entering it in memory, it is quicker to press the escape key than to backspace through the line buffer. An escape keystroke cancels the contents of the line buffer.

Although the editor is most useful for changing program lines, it can correct an immediate command before it is executed. It can also correct any user input required during a program run. The codes Basic/Debug recognizes for backspace and cancel are stored in the constant block and may be changed to support a special terminal or application. Section 7 describes how to alter the constant block.

Section 2

Elements of a Basic/Debug Expression

2.1 Introduction

Expressions represent the numeric values Basic/Debug needs to perform a task. An expression consists of one or more of the following elements:

- constants
- variables
- operators
- memory references
- function calls

The elements in a single expression are evaluated together when the statement is executed. The evaluation produces a single numeric value to be used in the execution of the instruction.

2.2 Number Handling

All calculations are performed in two eight-bit registers, require sixteen-bit values, and return sixteen-bit results. Basic/Debug adds a high-order byte of zeroes to any one-byte value before the calculation takes place. When a result exceeds sixteen bits, it is truncated and the excess significant bits are discarded.

All numeric values are internally represented in sixteen-bit binary two's complement form. In two's complement form, a negative number -n is represented by the bit pattern for 65536 - n. Therefore, a negative number has its high order bit turned on.

Numerical values range from -32768 to +32767. If a computation results in a value beyond the negative range, the answer is printed as a positive number. If a computation result is higher than the positive range, a negative number is printed. Table 2-1 shows examples of constants beyond the normal printing range of Basic/Debug.

Table 2-1. Basic/Debug Numeric Representation

Binary	Hex	Unsigned Decimal	Signed Decimal
0000 0000 0000 0000	0000	0	0
0000 0000 0000 0001	0001	1	1
1111 1111 1111 1111	FFFF	65535	-1
0111 1111 1111 1111	7FFF	32767	32767
1000 0000 0000 0000	8000	32768	-32768
1000 0000 0000 0001	8001	32769	-32767
0000 0001 0000 0000	0100	256	256
0000 0010 0000 0000	0200	512	512

Hexadecimal values are used frequently for addressing because hardware boundaries often occur on even hex addresses. Unsigned integers between 0 and 65536 may be entered to address memory locations. However, only values in the range of +32767 to -32768 are printed normally at the terminal. A method for printing values beyond the range is presented in Section 2.5.

2.3 Constants

A constant is a value that does not change during the program run and must be represented by a number. In Basic/Debug, a constant may be either a decimal or a hexadecimal value. The digits used to represent hexadecimal values are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The hexadecimal value A is equivalent to the decimal number 10. 10 in hexadecimal is equal to 16 in decimal. Basic/Debug requires a signal character, %, before a hexadecimal value. Any number not preceded by a percent sign is assumed to represent a decimal value.

A negative number is indicated by the minus sign, "-". Technically, this combines a constant with an operator to make an expression, and Basic/Debug treats it as such. This concept is important when omitting the PRINT keyword from a print statement (see Section 3.10).

Basic/Debug recognizes only whole numbers. Fractions cannot be entered, and the fractional part of any result is discarded. The following are examples of valid decimal and hexadecimal constants:

```
0          %
123        %7B
256        %100
32766      %7FFE
32768      %8000
```

2.4 Variables

A variable is a two-byte location where a numeric value may be stored. It is referenced by a single letter variable name. A variable may be changed or updated with a new value at any time.

Basic/Debug supports 26 variables. Each letter of the alphabet is used as a variable name. Variable storage is not cleared before a program is run, so it is possible to pass values from one program to another in variable storage.

The memory location of variable storage is fixed during the power-on-reset procedure of the Z8671 and depends upon the memory configuration available at that time. Variables are usually stored in the top page of RAM. Within the page, variables reside in locations 34-85. Two bytes are assigned to each variable. For example, variable A is stored in location 34-35, and variable Z is stored in location 84-85.

However, some Z8671 systems do not have any RAM. In this case, the variables are stored in Z8671 registers 34-85, which are shared with the GOSUB stack. In a No-RAM system, variables may be destroyed by the GOSUB stack. See Section 6.6 for a description of memory management in a No-RAM system.

2.5 Operators

An operator indicates a calculation to be perfomed when an expression is evaluated. Basic/Debug supports two sets of operators: arithmetic operators and relational operators.

Basic/Debug recognizes the following traditional operators for arithmetic functions:

+	addition
-	subtraction
*	multiplication
/	division

Operations are performed from left to right. If all four appear in a single expression, multiplication and division are performed first, followed by addition and subtraction. This may be altered by the use of parentheses. For example:

```
3*24-18/3+10 = 76
3*(24-18)/(3+10) = 1
```

Basic/Debug does not support fractional numbers, therefore, the remainder of the division in the second line is discarded.

A special division operator, the backslash "\", does unsigned division. It indicates that the dividend is to be treated as an unsigned integer in the range 0 - 65535. For example, the statement PRINT 40000\3 returns the correct answer, 13333. But PRINT 40000/3 returns -8512 because the signed integer for 40000 is -25536.

Because it treats the dividend as a sixteen bit positive number, the backslash can perform a logical right shift on a bit pattern, as shown below:

```
(-2)/2 = -1
(-2)\2 = 32767
```

An attempt to divide by a negative number with the backslash operator gives undefined results.

The backslash operator may also be used to print values higher than +32767. Assuming that N is a value out of normal printing range, the following statement will print N:

```
PRINT N\10; N-N\10*10
```

Relational operators specify conditional relationships in an IF statement. The six relational operators recognized by Basic/Debug are:

=	equal
<=	less than or equal
<	less than
<>	not equal
>	greater than
>=	greater than or equal

2.6 Memory References - Addresses

Basic/Debug can directly address the Z8671 internal registers and all external memory. The contents of any address may be examined and RAM may be altered. The location is specified by a memory reference, which has two parts: a signal character and an address value. A memory reference may be used in a Basic/Debug expression anywhere a variable may be used.

Any byte may be referenced by placing the byte signal character, "@", in front of the address. For example, @%1000 addresses the byte stored in location 4096. Byte references may be used to modify a single register in the CPU, control I/O devices, or access any memory location.

Sixteen-bit words are referenced with an address preceded by the word signal character "↑". This accesses the most significant byte at the address given plus the least significant byte at the next higher address. Modification of pointer register values requires a word reference.

The address value may be a variable, a constant, a hex value, an AND or USR function, an expression in parentheses, or, for indirect addressing, another memory reference. An expression is evaluated at run time and its value used as the memory address or register number to be referenced. For example, if the address needed depends on the value of C, Basic/Debug can perform the calculation:

```
145 LET @(C*100) = A
```

Indirect addressing can vector Basic/Debug through several addresses to find required information. For example,

```
PRINT ↑↑8
```

The first signal character "↑" indicates indirect addressing. Register R-8 is a sixteen bit pointer. It always contains the address of the first byte of the program in memory. To execute this instruction, Basic/Debug finds the address in R-8, then goes to that address and prints its contents. Pointer registers, which are discussed in detail in Section 6, require word references, as shown in the example above.

To modify Z8671 registers, use addresses in the range of either 0-127 or 240-255. The Z8671 has no registers implemented between 127 and 240. With this exception, Basic/Debug references registers between 00 and FF (hex), and external memory between 0100 and FFFF (hex). Do not use a word reference at address 00FF or FFFF because such a reference extends across internal/external memory boundaries, and returns a non-contiguous second byte.

Memory references may be used to implement arrays. Set aside a block of RAM to hold the array, and indicate the address of an element of the array by adding the element number to the array's starting address. For example, if an array of bytes starts at C000 hex, the following statements would define the starting address of the array and reference its elements:

```
A = %C000                 :REM ARRAY STARTING ADDRESS
@(A+J)=99                 :REM ELEMENT J = 99
@(A+I)=@(A+J)+@(A+K)      :REM A(I)=A(J)+A(K)
```

2.7 Functions

Basic/Debug supports two functions: AND, which performs a logical AND, and USR, which calls a machine language subroutine. These functions must be part of an expression. A function is treated as an operand, the same as a variable, constant, or memory reference. It does not change the order of arithmetic operations.

2.7.1 Logical Functions

AND performs a logical AND. It can be used to mask, turn off, or isolate bits.

```
AND (expression, expression)
```

The two expressions are evaluated, then their bit patterns are ANDed together. For example, AND (3,6) returns 2. If only one value is included in the parentheses, it is ANDed with itself.

To perform a logical OR, complement the AND function by subtracting each element from -1. For example, the function below is equivalent to the OR of A and B:

```
-1-AND(-1-A, -1-B)
```

The arithmetic sum may also be used for the logical OR operation if the bits to be added are known to be previously zero.

2.7.2 Machine Language Functions

An application often requires a subroutine which can be performed more quickly and efficiently in machine language than in Basic/Debug. The Z8 PLZ/ASM Language Manual (part number 03-3023-02) and the Z8 Assembler User Guide (part number 03-3048-02) describe the process of developing Z8 Assembly Language programs.

Basic/Debug can call a machine language subroutine which returns a value for further computation by the USR function. To call a subroutine which returns no value, use the GO@ command described in Section 3.2.

After the machine language subroutine is assembled, store it in memory that is not otherwise occupied by the Basic/Debug program or stack. The available memory space is indicated by the pointer registers described in Section 6. Use the address

of the first instruction of the subroutine as the first argument of the USR function, as follows:

```
USR (%2000)
```

Basic/Debug executes whatever it finds at this address. If there is no machine language routine at the location, the result is undefined.

The address may be followed by one or two values to be processed by the subroutine. For example:

```
USR(%2000,256,C)
```

The address and arguments are expressions separated by commas. Basic/Debug passes the values to the subroutine in registers R18-19 and R20-21, and expects the resulting value to be returned in R18-19. This resulting value is used to finish the evaluation of the expression.

The registers in which the arguments are passed depend on the number of arguments inside the parentheses. For example, the function USR(%700,A) calls the subroutine at %700 and passes it variable A in register R18-19. However, function USR(%700,A,B) passes A in R20-21 and B in R18-19. In either case, the machine language subroutine must leave the return value in R18-19.

Table 2-2. USR Arguments and Registers

call	R18-19 contains	R20-21 contains
USR (%700, A, B)	B	A
USR (%700, A)	A	A

The machine language subroutine must conform to the following requirements: it must end with a RET (hex AF) instruction, it must leave the value to be returned in R18-19, and it may use any of the free registers listed in the Memory Map in Section 6. The register pointer is set to point to R16-31 on entry to the routine, so the arguments may be fetched from working registers r_2-r_3 and r_4-r_5, and the return value left in r_2-r_3 (for a discussion of the working register feature of the Z8671 refer to the Z8 Microcomputer Preliminary Technical Manual, part number 03-3047-02).

2.8 Formal Syntax for Expressions

The syntax for Basic/Debug expressions is defined below in a meta-language descended from Backus-Naur form. The language follows the rules given at the beginning of Section 3.

```
expression      => [add_op] term (add_op term)*

signed_expression
                => add_op term (add_op term)*

add_op          => '+' | '-'

term            => factor (mult_op factor)*

mult_op         => '*' | '/' | '\'

factor          => variable
                => '@' factor
                => '↑' factor
                => number
                => '%' hex_number
                => AND '(' expression [',' expression] ')'
                => USR '(' address [',' arg1 [',' arg2]] ')'
                => '(' expression ')'
```

Section 3

Basic/Debug Statement Definitions

3.1 Introduction

Basic/Debug recognizes fifteen command keywords. The two most commonly used keywords, LET and PRINT, may be omitted when their arguments imply them. For example, a character string enclosed in quotation marks can only be processed by a PRINT command, so a quotation mark following a line number implies the PRINT keyword.

The first section of each of the following command descriptions defines command syntax. The meta-language used to define the syntax follows the rules below:

Syntactic constructs are denoted by lower case English words or phrases not enclosed in any special characters. Examples are command, stmnt, and gosub_stmnt.

The basic symbols of the language are keywords, written in upper case, and special characters, enclosed in quote marks. Examples are ',' LET '↑' NEW.

Possible repetition of a construct is indicated by appending either a '+' indicating one or more occurrences, or a '*', indicating zero or more occurrences. For example, the definition of number as digit+ means that a number consists of one or more digits.

Parentheses group a number of constructs together so that a repetition symbol (+ or *) may be applied to the group.

Square brackets denote optional items. The construct within the brackets may appear either zero or one time.

The vertical bar '|' signifies that one of several alternate constructs may be specified.

Curly brackets, '{' and '}', surround an English description of an otherwise indescribable construct.

The second section of each description lists sample statements which demonstrate the variety of commands possible within the syntax. The third section describes any special features of the command. The commands are listed in alphabetical order for easy reference. Statement syntax is summarized in Appendix A.

Syntax

```
go_stmnt => GO '@' address [',' arg_1 [',' arg_2]]

address  => expression

arg_1    => expression

arg_2    => expression
```

Examples:

```
GO@%E000, A, B
GO@%700
```

The GO@ command unconditionally branches to a machine language subroutine. It may only be used when the subroutine returns no value.

The first argument is the address of the first byte of the subroutine. The last two optional arguments are used to pass values to the subroutine. Unlike the USR function defined in Section 2.7.2, the contents of R18-19 are discarded and no value is returned. Otherwise, GO@ passes arguments to the subroutine in the same way USR does, according to the following table:

Table 3-1. GO Arguments and Registers

call	R18-19 contains	R20-21 contains
GO @%700, A, B	B	A
GO @%700, A	A	A

Syntax:

gosub_stmnt => GOSUB expression

Examples:

```
GOSUB 50
GOSUB C
GOSUB B*100
```

Often an application requires that a few lines of code be executed at several points in the program. Rather than repeat these lines at each location, isolate them at the beginning of the code. This subroutine may be called at any time during the program run by the GOSUB command.

Unlike Dartmouth Basic, the item following the keyword GOSUB may be either the number of the first line of subroutine or an expression which evaluates to the subroutine line number.

The subroutine must be terminated with a RETURN instruction. GOSUB stores the number of the next line to be executed where RETURN can find it to restart normal sequential execution. GOSUB must be the last instruction on its line.

One subroutine may call another. The RETURN instruction at the end of the second subroutine returns execution to the first subroutine. In this way, subroutines may be nested to the depth allowed by the memory available to the GOSUB stack.

Syntax:

goto_stmnt => GOTO expression

Examples:

```
GOTO 100
GOTO %FF
GOTO B*100
```

GOTO unconditionally changes the sequence of program execution. Unlike the Dartmouth Basic, Basic/Debug accepts expressions following the keyword GOTO. This feature allows a variable to be used to select a line number. For example, if the variable G will equal 1, 2 or 3 when line 100, 200 or 300 respectively is to be executed, use the following instruction:

```
GOTO G*100
```

GOTO is often used in the immediate mode for interactive debugging because GOTO enters the run mode. Unlike the RUN command, GOTO can specify the line number where execution is to begin. For example, when an error occurs and the following message appears at the terminal:

```
!ERROR AT LINE 4096
```

Line 4096 may be retried by entering the following command in the immediate mode:

```
GOTO 4096
```

Because GOTO unconditionally changes the sequence of execution, any statements that follow it on a program line can not be executed. Therefore GOTO must always be the last statement on a line.

Syntax:

```
if_stmnt => IF expression relational_op expression
              [THEN] apodosis

relational_op
         => '<' | '>' | '<>' | '=' | '<=' | '>='

apodosis => number | statement_line
```

Examples:

```
IF A>B THEN PRINT "A>B"
IF A>B "A>B"
IF X=Y IF Y=Z PRINT "X=Z"
IF A<>B I=0:J=K+2:GOTO 100
IF 1=2 THEN this part never matters
```

The IF/THEN command is used for conditional operations and branches. The apodosis may be another statement, a line number indicating another statement, or a list of statements separated by colons. Any of these statements may be another IF. The keyword THEN may be omitted to conserve memory space.

IF compares the value of the first expression to the value of the second. If the relationship indicated by the relational operator is true, then the apodosis of the instruction is executed. If the relationship is not true, then the next sequential instruction is executed.

There are only two conditions in which the keyword THEN may not be eliminated. It may not be omitted if the second expression ends with a decimal or hexadecimal constant and the line number of a statement is used. For example:

```
IF X <1 THEN 1000
```

The above statement requires a THEN to separate the numeric second expression from the line number. However, THEN may be eliminated from the statement by reordering the expressions:

```
IF 1 > X 1000
```

The second condition in which THEN may not be omitted is when the second expression ends with a hexadecimal constant, and the statement part is a LET statement in which the keyword has been omitted and the variable is between A and F. For example:

```
IF Z > %1000 THEN A = Z
```

No number of spaces in place of the THEN will prevent the interpretation of the variable letter as a hexadecimal value because spaces are ignored. THEN must be included to separate the expression from the apodosis.

Syntax:

```
input_stmnt => INPUT variable (',' variable)*
in_stmnt => IN variable (',' variable)*
```

Examples:

```
IN C, E, G
INPUT A
```

These statements first request information from the operator with the prompt "?", then read the input values from the keyboard and store them in the indicated variables. They are two of the three commands which assign an expression to a variable.

Either command accepts values for a list of one or more variables. If the user does not input as many values as are needed, both commands repeat the prompt until the required number of values are entered. The commands differ in the way they handle extra values entered by the operator.

INPUT discards any values remaining in the buffer from previous IN, INPUT, or RUN statements, and requests new data from the operator. IN uses any values left in the buffer first, then requests new data.

Unlike Dartmouth Basic, Basic/Debug accepts completely general expressions as input. It also accepts variables which have already been assigned a value. A variable assigned a value early in the list may be used to define a variable later in the list. For example, the statement INPUT C,A can process 10,C*5 as valid input.

When a program requires the operator to input a list of values, he may need to separate each item by a comma. Commas may be omitted if they are not needed to direct interpretation. Spaces are ignored. The following examples show how delimiters may be used to change the interpretation of input values:

```
? %123,A,ND(56)       (hex 123, variables A,N,D, decimal 56)
? %12 3AND(56)        (hex 123A, variables N, D, decimal 56)
? %123,AND(56)        (hex 123, value of 56 ANDed to itself)
```

Because Basic/Debug has only one input line buffer, INPUT and IN execute differently in the immediate and run mode. In the immediate mode, the user response overlays and destroys the INPUT or IN command that requested it. Consequently, no matter how many variables are listed after the keyword INPUT, only the first one is assigned to the input data.

However, IN may assign lists of variables and expressions in the immediate mode if both lists are alternately included in the command line. For example:

```
IN A, 10, B, 15, C, 20
```

When the above line is executed in the immediate mode, Basic/Debug fetches the first variable, A, from the keyboard buffer, and advances the buffer pointer. INPUT at this point would request a new input line from the keyboard, but IN, which uses all values in the buffer before issuing the "?" prompt, will return to the buffer and assign the value 10 to A. The process continues until all variables and values are used up. If the command line is closed with a variable, the "?" prompt is issued.

Generally, it is easier to use LET to assign values to variables in the immediate mode.

To help the operator enter the correct number and kinds of values, IN and INPUT are usually preceded by a PRINT statement describing the requirements. When the PRINT statement is terminated with a semicolon, the INPUT prompt "?" will be listed on the same line and appear to punctuate the message.

Although Basic/Debug does not support character string functions, the INPUT command may be used to accept a single letter as a user response, as shown below:

```
100 PRINT "PLEASE TYPE YES OR NO"
110 LET N=Y-1
120 PRINT "DO YOU UNDERSTAND";
130 INPUT N
140 IF N=Y THEN PRINT "GOOD!"
```

In this example, the value of Y does not matter. If the operator types Y, YES, YEAH, or YAH, then the variable N equals Y. If the operator type N, NO, or NOT YET, then variable N is unchanged and not equal to Y. To check for letters other than Y or N, use an unusual value for Y, such as -32323, and check both Y and Y+1 after input.

Syntax:

```
let_stmnt => [LET] left_part '=' expression`

left_part => variable | '@' factor | '↑' factor
```

Examples:

```
LET A = A+1
@ 1020 = 100
↑8 = %100*C
```

LET assigns the value of an expression to a variable or memory location. The left portion of the statement may be any alphabetic character A-Z, a memory reference, or a register reference. The value of the expression is either stored in the memory location, or placed in the variable's storage location, to be used at any subsequent appearance of the variable. Because the equal sign makes the syntax of this command unique, the LET keyword may be omitted.

A variable's value may be re-calculated by using the same variable on both sides of the LET assignment, as in the incrementing statement below:

```
LETB=B+1
```

LET may be used to store values in memory by using a memory reference on the left side of the LET assignment, as shown below:

```
LET@1024=B/2
```

When this statement is executed, the memory reference is calculated first, then the expression is evaluated and its value stored. A word memory reference stores the most significant byte in the location addressed. The least significant byte is stored in the next higher address. Take care when modifying internal registers or the area where the program is stored in memory because improper changes could have catastrophic results.

Syntax:

```
list_stmnt     => LIST [starting_line[',' ending_line]]

starting_line => expression

ending_line    => expression
```

Examples:

```
LIST
LIST 200, 1000
```

This command is used in the interactive mode to generate a listing of program lines stored in memory on the terminal device. The optional line numbers specify the range of lines to be listed. If only one number is given, only that line will be listed. If ending_line is included, only starting_line through ending_line inclusive will be listed. A LIST command without arguments lists all the lines in the program.

The LIST command is generally used in the immediate mode, however, it may be used in the run mode for simple text processing. Because Basic/Debug does not examine program lines after the line number until runtime, it can process text, as shown in the following program:

```
100 REM THIS PROGRAM PRINTS A MESSAGE N TIMES
110 IF N>0 THEN 200
120 : PRINT "HOW MANY TIMES";
130 : INPUT N
200 REM BEGIN LOOP
210 : LET N=N-1
220 : LIST 1000, 1070
230 : IF N>0 THEN 210
240 STOP

1000|    This is a message saved in memory.  It will be
1010|printed when the program is RUN.  If you tried to
1020|execute lines 1000 to 1070 you would get an error
1030|message.  But in this program, lines 1000+ are not
1040|executed, just LISTed.
1050|
1060|                          (Signed)
1070|
```

Five lines of this program are indented to show program structure and make it easy to read. The colon prevents Basic/Debug from removing the spaces before the statement portion of the line. When the program is executed, the message will be printed exactly as it appears in lines 1000-1070, including the vertical bars along the left edge. The vertical bar is needed to indent line 1000; the others are included for consistency. In summary, use a colon to indent an instruction because Basic/Debug recognizes it as a statement delimiter, and use the vertical bar to indent text lines because it is the least distracting character to have printed down the left side of a page.

Syntax:

new_stmnt => NEW

Example:

NEW

The NEW command resets pointer R10-11 to the beginning of user memory, thereby marking the space as empty and ready to store a new program. If this command is entered in error, take heart, the stored program is not really gone. Although it may not be modified, it may at least be listed by setting the line number of the first line back to a very small number. Use a LET statement in the immediate mode, as shown in the example below:

```
LET ↑↑8=1
```

Although an attempt to run the program after this kind of recovery may appear to work, there is no longer any memory over-run protection, and the program may be destroyed.

Syntax:

```
print_stmnt  => PRINT [item (delimiter item)*] [delimiter]
             => initial item (delimiter item)* [delimiter]

delimiter    => ',' | ';'

item         => quoted_string | expression | HEX '('expression')'

initial_item
             => quoted_string | signed_expression | '('expression')'

quoted_string
             => '"' { any character sequence, not containing
                nulls, deletes, linefeeds, carriage returns,
                escapes, backspaces or quotes}'"'
```

Examples:

```
PRINT HEX (255)
"THE ANSWER IS ";X
(A*100)
+%800 + Z
PRINT A, B, C, D, E
```

The PRINT command lists its arguments, which may be text messages or numerical values, on the output terminal. The delimiters used in the argument specify how the items are to be printed on the screen.

Characters and spaces enclosed in quotation marks are listed exactly as they are typed. Quotation marks are unprintable. If a message must be punctuated with a quotation mark, use the single quote or apostrophe instead. As mentioned above, a character string enclosed in quotation marks implies the PRINT command, so the keyword may be omitted. The PRINT keyword without an argument or terminating delimiter generates a blank line. Any PRINT instruction can be followed by a colon and another statement.

When an expression is entered as the argument to the PRINT command, Basic/Debug evaluates it and lists its decimal value at the terminal. Only the significant digits are printed; leading zeros and divisional remainders are not. PRINT treats numbers as signed integers. A method for printing unsigned values is presented in Section 2.5.

To PRINT a hexadecimal value, use the syntax:

```
PRINT HEX (expression)
```

Basic/Debug evaluates the expression, and prints its positive hexadecimal equivalent. The PRINT command cannot list a negative hexadecimal number.

Unlike character strings, the HEX function must be preceded by the PRINT keyword, as must any expression beginning with a variable. However, the keyword may be omitted before an expression if the expression is preceded by a "+" or "-". For example, -10 + 20 or +20 - 10 entered as statements print a value of 10, but 20 - 10 results in an error message.

When a comma is used to delimit items in a PRINT statement, a tab is generated between each item. The tab stops are located at eight-space intervals across the screen. To print left-justified columns, simply put all the items to be printed on one line in one PRINT statement, and separate them by commas. The first character of the data item will appear in the column containing the tab stop. If the item is longer than eight characters, Basic/Debug tabs to the next available stop to print the next item.

To print one item directly after another without any spacing, use a semicolon as a delimiter. For example, the command:

```
PRINT"OUTPUT=";X
```

will print the value of variable X immediately after the equal sign. If a PRINT statement is ended by a semicolon, no carriage-return-linefeed is generated. The next item printed by a subsequent statement will appear on the same line as the item that preceded the semicolon. A comma at the end of a PRINT statement will also suppress the carriage return sequence, however, the next item to be printed appears at the next eight column tab stop.

To print right-justified columns, as is necessary with lists of figures, leading spaces must be added. Basic/Debug can only print spaces enclosed in quotation marks. The following example program adds leading spaces to N:

```
200 IF N<10000 THEN PRINT " ";
210 IF N<1000 THEN PRINT " ";
220 IF N<100 THEN PRINT " ";
230 IF N<10 THEN PRINT " ";
240 PRINT N
```

Basic/Debug can print most control characters such as the bell if they are contained in a quoted character string. The following control characters cannot be printed:

back space	(^H)
escape or cancel	(ESC)
carriage return	(CR)
linefeed	(LF)
delete	(DEL)
null	(NUL)

The circumflex, "^", indicates that the control key is held down while the specified key is pressed.

Although control characters can be printed, most terminals do not advance the cursor when the control character is received. Therefore, Basic/Debug's cursor pointer may not indicate the true position of the cursor after control characters are printed. When this is the case, any attempt to print in columns using the comma delimiter will fail. For example:

```
 5 X=0
10 PRINT "X^G", X
20 PRINT "X", X
```

When the above program is executed, the following output appears at the terminal:

```
X       0
X        0
```

The statement at line 10 inserts only seven spaces because the control-G character pushed the cursor pointer one place to the right of the cursor's actual position.

Syntax:

```
rem_stmnt => REM {all following characters to the end
                  of the line}
```

Examples:

```
REM  CONTROL LOOP
REM  SUBROUTINE NAME
REM  CODE EXPLANATION
```

The REM command is used to insert comments, remarks, or other explanatory messages into the code. Basic/Debug ignores anything following the REM keyword, therefore, REM and its comment must be the last command on a line. The liberal use of remarks throughout a program makes it easier to read and maintain. However, remarks take space in memory, and should be omitted for maximum space utilization.

Syntax:

```
return_stmnt => RETURN | RET
```

Examples:

```
RETURN
RET
```

RETURN is always the last instruction of a subroutine, and may be abbreviated as RET. It does not require an argument because GOSUB stores the next line number where RETURN can find it to restart normal sequential execution. RETURN must be the last instruction on a line.

If one subroutine calls another, the RETURN instruction at the end of the second subroutine returns execution to the first subroutine. In this way, subroutines may be nested to the depth allowed by the memory available to the GOSUB stack.

Syntax:

```
run_stmnt => RUN [expression (',' expression)*]
```

Examples:

```
RUN
RUN 17, %200, 23
```

This command initiates sequential execution of all instructions stored in memory. RUN is used only in the immediate mode. Data values for the first IN command may be entered, separated by commas, between the keyword RUN and the terminating carriage return, as shown below:

```
RUN 45,-583
```

Syntax:

```
stop_stmnt => STOP
```

Example:

```
STOP
```

STOP gracefully ends program execution, and clears the GOSUB stack. A STOP instruction is implied after the last program line in memory, so a terminating STOP command may be omitted from the program to save memory space.

Program execution is often ended abruptly by an error. After altering the errant statement in the immediate mode, the user may restart the run by using GOTO with the appropriate line number, or reset the program with a STOP instruction, and RUN the program again from the beginning.

Section 4

Errors and Interactive Debugging

4.1 Errors

Errors occur when Basic/Debug is unable to interpret an instruction. An error returns the system to the immediate mode with all variables and the GOSUB stack unaltered and sends the user an error message. Error messages appear at the terminal in the following format:

```
!error_code AT line_number
```

The numerical error codes are defined in Appendix C. If the error is found while a program is running, the error message will contain a line number. An error may occur in the immediate mode, but no line number will be listed. An error occurs when the keyword or argument is unrecognizable or unexecutable, or, in the case of an IN or INPUT instruction, that the data input by the operator is unintelligible. A control G is sent to the terminal with the error message to ring the terminal bell.

4.2 Interactive Debugging

Basic/Debug allows interruptions and changes during a program run to correct errors and add new instructions without disturbing the sequential execution of the program.

To interrupt a running program, press the escape key. Escape causes error 0 at the next line to be executed. The GOSUB stack and variables are preserved, so that after the program is edited, the run may be restarted by a GOTO command with the appropriate line number. Use the same procedure to correct code where Basic/Debug returns an error. Registers and the GOSUB stack are maintained through any error condition, so a problem line may be retried any number of times.

Escape is tested when execution advances sequentially from one line to the next. It is not tested between statements on a single line, or after a GOTO or GOSUB. It is tested after a RETURN. Likewise, all conditional instructions following an IF command are executed before the escape condition is tested.

If execution is caught in a loop in a machine language subroutine or in a one line GOTO loop, an escape keystroke cannot interrupt the program. In this case, execution must be terminated by a CPU reset. The program may be preserved in memory during reset by holding down the break key while pressing the reset button.

Escape also cannot interrupt a program during the execution of an INPUT or IN statement. Escape merely cancels the input line buffer and reissues the prompt "?". So, to generate an error, enter something unintelligible, such as a period.

To cause a break in a program, enter a line number followed by something uninterpretable, again for example, a period. Do not use STOP in a program to be debugged interactively because its execution clears the GOSUB stack.

Section 5

Execution Speed vs Memory Space

5.1 Introduction

Depending on the application, some programs are limited by the memory space available, while others are limited by the time necessary to execute each instruction. There are trade-off relationships between program readability, memory space, and execution time. The following sections advise how to code a program for either minimum memory usage or minimum execution time.

5.2 Conserving Memory Space

To conserve memory space, eliminate all optional keywords such as LET, PRINT, and THEN from the code wherever possible. Abbreviate RETURN to RET and eliminate all spaces. A STOP command at the end of the program is implied, and so it may be omitted. Remarks should also be deleted.

There are three ways to conserve memory space by reworking line numbers. First, although most program line numbers are stored in binary, those used as arguments for GOSUB and GOTO are stored in ASCII, one byte per digit. Bytes are saved by using low line numbers for the destination subroutine.

Second, space is allotted for 26 variables whether they are used or not. Variables not used by the program may be used to address subroutines or GOTO destinations by storing the line number in the variable. For example, if the subroutine at line 4000 is used several times in a program, the following statements store the line number in the variable, then use the variable as a line number:

```
LET A = 4000
GOSUB A
```

The variable A could be used elsewhere in the program where the value 4000 is needed as data; Basic/Debug makes no distinction between data and line number values.

Third, each line has an overhead of three bytes in memory: two for the line number and one for the terminating null. This overhead may be reduced to one byte per statement by entering several statements on a single line. The overhead is then one byte for the colon separating the statements.

5.3 Improving Execution Time

Although Basic/Debug does not execute instructions as quickly as machine language code, some coding practices improve execution time. Including the keyword LET, for example, eliminates the interpreter's search through the keyword list to find the implied command. Including the keyword PRINT before an expression also eliminates the search process, but a print statement beginning with a quotation mark executes more quickly than one beginning with the PRINT keyword.

Speed of execution may also be improved by eliminating spaces, remarks and THEN, and also by abbreviating RETURN as RET. It takes longer to convert two or more ASCII digits to binary than to fetch a variable from memory. Use variables for any frequently needed large constant.

Using low line numbers for frequently used subroutines saves execution time as well as memory space. When normal sequential execution is interrupted by a GOTO, GOSUB, or RETURN, Basic/Debug scans the program from the beginning until it finds the desired line. Therefore, the closer the desired line is to the beginning of the program, the sooner the search succeeds.

If a choice is allowed for arithmetic instructions, use the fastest executing operation possible. Multiplication executes more quickly than addition, which is faster than subtraction or division. Unnecessary parentheses also slow execution and should be omitted.

To find out which sections of a program execute slowly and need improvement, use the Z8671 internal timer T1 to measure the execution speed of different portions of the programs. First, initialize the timer to run at its slowest rate:

```
LET @ 243 = 3
LET @ 242 = 0
LET @ 241 = 14
```

Then, around each statement or group of statements to be timed, insert these instructions to add up the amount of time spent executing:

```
100 Z = 242: Y = @Z
101 REM - these are the statements to be timed
102 Z = @ Z:S = S + AND(Y-Z-97, 255)
```

S is the running sum for time spent on the statement. 97 is an offset representing the measuring overhead, so that S is not changed if line 101 is omitted. If a section of the program to be timed takes longer than 256 counts on T1, it will overflow, so add an extra offset of 256 or 512 or whatever is needed.

A program may be timed by T1 without inserting extra statements, but the machine language routine required for the operation is beyond the scope of this manual.

Section 6

The Memory Environment

6.1 Memory Structure

The Z8671 system uses three kinds of memory: registers, internal ROM and external ROM or RAM. Basic/Debug assigns addresses 0 through 255 to the register file. The 144 registers include four I/O port registers (R0-R3), 124 general purpose registers, and sixteen control and status registers. The port, control, and status registers are common to all Z8 Family CPU chips and are described in the Z8 Technical Manual. However, Basic/Debug uses many of the general purpose registers as pointers, scratch workspace, and internal variables. So, in a Z8671 Basic/Debug system, these registers cannot be used by a machine language subroutine or other user programs.

The 2K of internal ROM on the Z8671 chip contains the Basic/Debug intepreter. It begins at address 00 and extends up to 2047, but because Basic/Debug assigns the addresses 0 - 255 to the register file, the lower 256 bytes of internal ROM may be accessed only by machine language instructions.

The Z8671 is configured on power-on/reset by Basic/Debug for external program memory. External memory can be reconfigured after power-on/reset to meet the needs of the applicaion and can use all the external memory features available with the Z8 Family of Microcomputers. The details of external memory configuration programmable options are described in the Z8 Family of Microcomputers Product Specification, part number 00-2037-A. The external memory space can be populated with a combination of ROM, RAM, and I/O.

6.2 Initialization and Automatic Start-Up

On power-on/reset, Basic/Debug sizes RAM memory and checks for an auto start-up program. Basic/Debug nondestructively tests memory from low to high addresses. Only one byte of every 256 is tested at relative location xxFD (hex). The first byte of RAM found determines the low boundary of user memory. Basic/Debug assumes that if xxFD is RAM, xx00 is also RAM and sets pointer register R8-9 to xx00 (hex). Basic/Debug continues to test up through memory until it finds a byte that does not contain RAM. Basic/Debug assumes it has RAM up to and including yyFF (hex) where yyFD is the last location tested that contained RAM. The top of user memory pointer, R4-5, is set to yy20.

Automatic start-up allows a program stored in ROM to be executed on reset without operator intervention. Automatic execution occurs on power-up/reset when the program:

- is stored in ROM
- begins at 1020 (hex)
- begins with a line number between 1 and 254 inclusive

6.3 Program Format

Program lines are stored in memory in the following format:

```
. 06 . 60 . L . I . S . T . 0 .

  line number        statement in ASCII       null
   in binary
```

The end of the program in memory is marked by FFFF (hex), which is equivalent to the illegal line number 65535 and causes Basic/Debug to stop. Below is a sample program and a memory dump of that program:

```
10 I=5
20 PRINT "WELCOME TO BASIC/DEBUG"
30 I=I-1: IF I
40 STOP
```

```
00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00    *................*
00 00 00 00  00 00 00 00  00 00 00 C0  00 00 00 00    *................*
00 0A 49 3D  35 00 00 14  50 52 49 4E  54 20 22 57    *..I=5...PRINT "W*
45 4C 43 4F  4D 45 20 54  4F 20 5A 38  20 42 41 53    *ELCOME TO Z8 BAS*
49 43 2F 44  45 42 55 47  22 00 00 1E  49 3D 49 2D    *IC/DEBUG"...I=I-*
31 3A 20 49  46 20 49 3E  30 20 47 4F  54 4F 20 32    *1: IF I>0 GOTO 2*
30 00 00 28  53 54 4F 50  00 FF FF 00  00 00 00 00    *0..(STOP........*
00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00    *................*
```

6.4 The Top Page of RAM

When Basic/Debug discovers RAM in the system, it initializes the pointer registers to mark the boundaries between areas of memory that are assigned specific uses. The top page of RAM is allocated for the line buffer, variable storage, and the GOSUB stack. Table 6-1 gives the relative addresses of the elements contained in the top page of RAM.

In the following tables, yy signifies the high-order byte of the address of the last page of RAM, as contained in pointer register R10.

Table 6-1. The Top Page of RAM

ADDRESS (hex)	CONTENTS
yyF1 - yyFF	Unused.
yy68 - yyF0	Input line buffer, used for editing in immediate mode and user response to IN or INPUT request in run mode.
yy56 - yy67	Unused.
yy54 - yy55	Storage for variable Z.
yy52 - yy53	Storage for variable Y.
. . .	
yy22 - yy21	Storage for variable A.
yy20	Base of GOSUB stack. Stack grows down to lower memory addresses, and may extend until it reaches the top of the user's Basic/Debug program.

6.5 Pointer Registers - RAM System

A pointer register is two eight-bit registers. Two pointers indicate the current contents of the input line buffer. Four other pointers manage user memory, as shown in Table 6-2.

Table 6-2. Memory and Pointers

ADDRESS (hex)	CONTENTS (hex)	DESCRIPTION
R0E-0F =>	yy68 to yyF0	Next value to be used from line buffer. INPUT command resets to the beginning of the buffer; IN uses all values in the buffer before resetting.
R0C-0D => (R12-R13)	yy68 to yyF0	Last character entered in line buffer. Backspace subtracts one from this pointer; escape resets it to the beginning of the buffer. R12 is the page number for variables and the input buffer.
R0A-0B =>	yy20	Top of user memory, high boundary of GOSUB stack. Initially set to yy20 of high page of RAM.
R06-07 =>	moves down from yy20	Low boundary and top of GOSUB stack.
R04-05 =>	moves up from xx00	High boundary of user program plus stack reserve.
R08-09 =>	xx00	Bottom of user memory; first line of user program.

Pointers R4-5 and R6-7 track the progress of the GOSUB stack and the user program as they grow towards each other. R4-5 marks the top of the user program, plus a reserve that buffers any potential collision with the stack. The NEW command sets this pointer back to the beginning of the user memory as indicated by pointer R8-9, plus the stack reserve indicated in the control block. The default size of the stack reserve is 32 bytes, which should be enough for normal operations. However, extended machine language programming or deeply nested interrupts may require an increased reserve. Section 7.1 contains instructions for altering the control block.

Pointer R6-7 marks the lowest address used for GOSUB, which is the top of the stack. It is also the base of the machine-language/interrupt stack. These elements build down from the GOSUB stack towards the program. However, because

there is no pointer register for the top of the machine language stack, no error is generated if it overflows into program area. An overflow error is only signaled when R6-R7 crosses R8-9. This is why there is a stack reserve at the end of the user program, and why an expanded reserve may be necessary for extensive machine language subroutines and interrupts.

Occasionally a page or more of memory may be needed for a special application such as an I/O buffer. Once R8-9, R10-11, and R12-13 are set during the intialization procedure, Basic/Debug never alters them. Therefore, areas of memory may be reserved by changing these pointers with a LET statement. For example, to save a block at the bottom of user memory, add the desired length of the block to R8-9. The user program is stored beginning at the address stored in R8-9, leaving the lower addresses for the application. To reserve an area between the GOSUB stack and the variables, move R10-11 down by the desired amount.

To reserve a space above the variables, move R12 down. R12 must be moved in multiples of 256 bytes. Changing R12 also redefines all the variables so that none of them contain their previous value. This characteristic may be used to make more variables available, if used with caution. Whenever R12 is changed, R10 must also be moved to keep the GOSUB stack clear of the variables.

When the pointers at R8 or R10 are changed, a NEW command must be entered to initialize all the other pointers that depend on the altered values.

6.6 Register Management for a No-RAM System

When Basic/Debug tests the available memory and finds no RAM, it uses an internal stack and shares register space with the input line buffer and variables. This limits the depth of the GOSUB stack, the length of the line buffer, and the number of usable variables. Because there is no external memory, some pointer registers become meaningless.

Table 6-3 maps the contents of the Z8671 registers when there is no RAM in the Z8671 system.

Table 6-3. Register Map for No-RAM System

ADDRESS (hex)	(decimal)	CONTENTS
F0-FF	240-255	Z8671 control registers. See map for RAM system.
80-EF	128-239	No registers are implemented at these addresses.
68-7F		The Expression Evaluation stack grows from 7F (hex) down, and the line buffer grows from 68 (hex) up.
40-67	64-103	GOSUB stack; grows down.
40-55	64-85	Area shared by variables M-Z and GOSUB stack. Variables are destroyed if stack grows into this range.
22-55	34-85	Variables A through Z.
21	33	Free register, available for USR subroutine.
20	32	Print column counter, contains current cursor location.
1F	31	Internal variable. Do not modify.
1E	30	Basic/Debug uses as scratch. USR subroutine may use, but cannot save values here.
1C-1D	28-29	Pointer to constant block.
18-1B	24-27	Internal variables. Do not modify.
16-17	22-23	Current line number.
14-15	20-21	Second argument in three argument USR subroutine call.
12-13	18-19	Last argument and result in USR subroutine call.
10-11	16-17	Basic/Debug uses as scratch. USR subroutine may use, but can not save values here.

Table 6-3. Register Map for No-RAM System (cont.)

ADDRESS (hex)	(decimal)	CONTENTS
0E-0F	14-15	Pointer to next character to be used in input buffer.
0C-0D	12-13	Pointer to the end of the line buffer. R12 defines the page containing the variables, and so contains 00 to indicate the registers.
0A-0B	10-11	Pointer to bottom of GOSUB stack. Initialized to 68 (hex).
08-09	8-9	Pointer to start of Basic program. This address will be in external ROM, usually 1020 (hex) for auto-start up.
06-07	6-7	Pointer to top of GOSUB stack. Since stack is in registers, 00 will be in R6 and the register number in R7.
04-05	4-5	Free register. Available for USR subroutines.
00-03	0-3	Z8671 I/O ports.

The pointer registers have the register number in the least significant or odd-numbered byte, and the page number 00 in the most significant byte.

The GOSUB stack starts at R-103 and grows to R-58 before signaling a stack overflow. By the time the overflow occurs, variables M-Z will already be destroyed. If deeply nested subroutines are used, it is safest to use only the first eight variables, A - H. However, even these can be destroyed by machine language code. Of course, if the program contains only simple expressions and few subroutines, all the variables are available. To test if variables are being destroyed, set the last variable used to a known value, run the program, and see if it changes.

6.7 The Memory Map

Table 6-4 gives the complete memory map for a Z8671 Basic/Debug system with RAM.

Table 6-4. Memory Map

ADDRESS (hex)	(decimal)	CONTENTS
FFFD	65533	Baud Rate. Pattern of lowest three bits sets baud rate. See Appendix B.
xxFF		Top of External Memory.
xx00		Bottom of highest memory page.
1020	4128	First address of Basic/Debug program for auto-start up. Store program starting in byte 20.
1015	4117	Address for external output driver. Store a jump to a user-supplied output driver here.
1012	4114	Address for external input driver. Store a jump to a user-supplied output driver here.
100F	4111	Address for IRQ5. When an interrupt occurs at IRQ5, the Z8671 vectors through internal ROM to 100F to get a jump to the address of a user-supplied processing routine.
100C	4108	Jump to routine for IRQ4.
1009	4105	Jump to routine for IRQ3.
1006	4102	Jump to routine for IRQ2.
1003	4099	Jump to routine for IRQ1.
1000	4096	Jump to routine for IRQ0.
0800	2048	Bottom of external memory.
07FF	2047	Top of Z8671's internal ROM.

Table 6-4. The Memory Map (cont.)

ADDRESS (hex)	(decimal)	CONTENTS
07FF-00	2047-0	Basic/Debug Interpreter.
00-FF	00-255	ROM unaddressable by @ or ↑.
20-27	32-39	Default constant block.
00-0B	00-11	Interrupt Request Vectors, start of Z8671's internal ROM.
THE REGISTER FILE		
ADDRESS (hex)	(decimal)	CONTENTS
00FF	255	Stack Pointer (Bits 7-0)
00FE	254	Stack Pointer (Bits 15-8)
00FD	253	Register Pointer
00FC	252	Program Control Flags
00FB	251	Interrupt Mask Register
00FA	250	Interrupt Request Register
00F9	249	Interrupt Priority Register
00F8	248	Ports 0-1 mode
00F7	247	Port 3 Mode
00F6	246	Port 2 Mode
00F5	245	T0 Prescaler
00F4	244	Timer/Counter 0
00F3	243	T1 Prescaler
00F2	242	Timer Counter 1
00F1	241	Timer Mode
00F0	240	Serial I/O

Table 6-4. Memory Map (cont.)

ADDRESS (hex)	(decimal)	CONTENTS
80-EF	128-239	No registers implemented.
40-7F	64-127	Expression Evaluation Stack. Basic/Debug uses all these registers only for the most complicated expressions. Usually the lower registers are available for USR subroutines.
21-3F	33-64	Free registers available for USR subroutines. They are never used by Basic/Debug interpreter.
20	32	Print column counter, contains current cursor location.
1F	31	Stores internal variables used by interpreter. Do not modify.
1E	30	Interpreter uses as scratch. USR may use, but cannot save values here.
1C-1D	28-29	Pointer to the constant block.
18-1B	24-27	Stores internal variables used by interpreter. Do not modify.
16-17	22-23	Contains line number currently being executed.
14-15	20-21	Passes second argument to USR subroutine in three-argument call.
12-13	18-19	Passes last argument and result of USR subroutine call.
10-11	16-17	Interpreter uses as scratch. May be used in USR subroutine, but value cannot be saved.
0E-0F	14-15	Pointer to next unused character in the input line buffer.

Table 6-4. Memory Map (cont.)

ADDRESS (hex)	(decimal)	CONTENTS
0C-0D	12-13	Pointer to end of line buffer. Indicates last character entered in immediate mode or in response to a "?" prompt in the run mode. R12 (0C) contains the number of the highest page in RAM, or 00 if there is no RAM.
0A-0B	10-11	R10-R11. Pointer to stack bottom, high boundary of usable memory. The Z8671 stack pointer is initialized to the value contained in this pointer. On powerup, these are initially set to xx20 of the highest page of RAM.
08-09	8-9	R8-9. Contains address of the first byte of a user program. Also points to the lowest byte of user memory, initially set to first location in external RAM, or to location 1020 if there is ROM there.
06-07	6-7	R6-7. Marks the top of the GOSUB stack, also the base of the machine language stack. It pushes down two bytes per GOSUB, and pops up two on each RETURN. If this pointer crosses the program end pointer (R4-5), a memory overflow error occurs and Basic/Debug aborts the program or the insertion that caused it. The stack reserve at the end of the program prevents any damage to the code. NEW and STOP statements reset this pointer to the value in R10-11.

Table 6-4. Memory Map (cont.)

ADDRESS (hex)	(decimal)	CONTENTS
04-05	4-5	R4-5. Pointer to end of Basic/Debug program in RAM, plus a stack reserve. If the GOSUB stack pointer reaches this address, an overflow error results. The NEW command sets this to the value of R8-9 plus the stack reserve, a value which is set by the fifth byte in the constant block. As program lines are entered, this pointer advances to protect the program. Program data stored above this pointer may be destroyed by deeply nested subroutines or by program modification. However, Basic/Debug keeps this pointer a safe distance ahead of the last program line so that if any overflow does occur, the program is not affected.
00-03	0-3	Z8671 I/O Ports

Section 7

The Constant Block, Interrupts, and I/O Drivers

7.1 The Constant Block

The constant block can reconfigure the Z8671 Basic/Debug system to support unusual terminal characteristics, and expand the stack reserve in user memory. The default constant block is located in the bottom page of internal ROM and can be accessed only by machine language code. However, Basic/Debug can be directed to use a user-supplied constant block stored in external memory.

The Z8671 Basic/Debug system comes configured for a standard ASCII CRT terminal or teleprinter. Because Basic/Debug echoes all input back to the terminal, the terminal must operate in full-duplex mode, and display or print only what is output from the Z8671. Basic/Debug outputs a simple carriage-return line feed sequence (hex OD, OA) with no pad characters as a line separator. It uses ESC (hex 1B) as the line cancel code and BS (hex 08) as backspace.

To reconfigure Basic/Debug to support alternate codes, store a 16 byte constant block in external ROM or RAM memory at an address divisible by 16. The following table summarizes the contents of the constant block:

Table 7-1. The Constant Block

Relative address (hex)	Contents	Default
xxx0	%06	%06
xxx1	%50	%50
xxx2	stack reserve	%20
xxx3	backspace code	%08
xxx4	cancel code	%1B
xxx5 to xxxF	line separator sequence terminating with %FE	%OD OA FE

The first two bytes of the constant block must contain hex 0650. The third byte of the constant block sets the size of the stack reserve at the end of the user program in memory. The default value is 32 (hex 20), but an application using deeply nested interrupts and many machine language subroutines may require a larger reserve.

Basic/Debug compares every input character to the next two bytes of the constant block because they are editing codes. The code in the fourth byte causes Basic/Debug to adjust buffer pointer R12-R13 to delete the previous character. Nothing is echoed except the input character.

In immediate mode, the code in the fifth byte causes Basic/Debug to cancel the content of the line buffer. When an input character matches the content of xxx4, the pointer is reset to the beginning of the line buffer. In run mode, this code escapes the program by returning error 0.

The last ten bytes of the block define the line separator sequence. However, a maximum of only nine bytes can be sent to the terminal at the end of each line because the line separator must be followed by a hexadecimal FE before the sixteenth byte of the block. If a terminal requires more than seven pad characters, a special data rate or other unusual format, alternate I/O drivers must be supplied. Instructions for using external I/O drivers are given in Section 7.3.

To initialize the constant block, place its address in the constant block pointer register R28-29. Use a simple LET statement, for example, if the block is at 2000-200F (hex):

```
LET ↑ 28 = % 2000
```

When the constant block and an auto-start program are stored in ROM, include a statement early in the program to initialize the constant block.

7.2 Interrupts

Basic/Debug does not process interrupts. Interrupts are vectored through locations in internal ROM which point to addresses 1000-1011 (hex). To process interrupts, put jump instructions to the interrupt handling routines at the appropriate addresses as shown in Table 7-2.

Table 7-2. Interrupt Jump Instructions

Address (hex)	Contains Jump Instruction and Subroutine Address for:
1000-1002	IRQ0
1003-1005	IRQ1
1006-1008	IRQ2
1009-100B	IRQ3
100C-100E	IRQ4
100F-1011	IRQ5

Basic/Debug uses the internal UART for all program and data I/O. Applications programs may use other I/O, but independently of Basic/Debug.

7.3 I/O Drivers

Basic/Debug operates the UART in a polling mode, waiting on each input and output character. In normal operation, input is accepted one line at a time. Output is printed similarly in character groups. Therefore, Basic/Debug alternately computes and then waits on the I/O process. To increase throughput, these periods of computing and waiting may be overlapped by external buffered I/O drivers.

External, user-supplied I/O drivers can increase throughput by buffering, use devices other than the internal UART, add more than seven pad characters to the end of each line, or perform any other application-specific I/O tasks.

Each time Basic/Debug needs a new input character, it checks the Interrupt Mask register to see if the serial input interrupt (IRQ3) is disabled. If so, it jumps to the internal I/O input driver. If not, it goes to external memory location 1012 (hex) for a jump instruction to a user-supplied input driver. Character output is similarly controlled, but the jump instruction to the external output driver is contained at 1015 (hex).

Table 7-3 is based on the bits of the Interrupt Mask Register, shown in Figure 7-1. Bit seven enables interrupts. Bit three masks IRQ3, which then enables the internal I/O drivers.

Table 7-3. External I/O Driver Conditions

Interrupts Enabled? (D7 in IMR = 1)	IRQ3 Enabled? (D3 in IMR = 1)	Go to External I/O Driver
yes	yes	yes
yes	no	no
no	yes	yes
no	no	yes

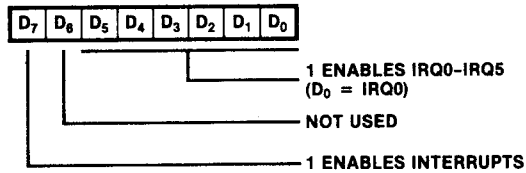

Figure 7-1. Interrupt Mask Register

External I/O drivers must conform to the following requirements: They must pass a single ASCII input or output character in R19 with the register pointer set to R16-31. Registers R4-R15 and R22-32 must be preserved. Location %1012 must contain a jump to the character input driver, and %1015 a jump to the character output driver. The input routine must do any necessary echoing, and the parity bit (bit 7) of all input characters must be set to 0. Drivers return to Basic/Debug with a RETURN instruction (hex AF).

Whether or not external I/O drivers are enabled, though it is of primary use when they are, the program execution can be halted by setting user flag 1, which is the least significant bit of the flag register (R252), to 1. Normally an interrupt-driven input routine would be looking for an escape character in the incoming data; if one comes up, the low bit in the flag register would be set to one. Note that user flag 2 should not be altered by the I/O routines.

7.4 Binary I/O

Basic/Debug programs may read and write binary data by directly calling the single-character input/output drivers with a USR or GO@ statement.

Input driver address:	%54
Output driver address:	%61

The routines at the indicated addresses will either use the built-in UART drivers, or jump to external, user-supplied routines, depending on the conditions discussed in Section 7.3.

The following example program prints the hex equivalent of an ASCII character.

```
10 PRINT "INPUT A CHARACTER, PLEASE";
20 C = USER (%54)
30 PRINT "  THE HEX VALUE OF ";
40 GO@ %61, C
50 PRINT " IS "; HEX (C);".  SHALL WE DO ANOTHER?";
60 Q = USER (%54)
70 PRINT : IF Q = %59 GOTO 10
80 REM %59 IS AN ASCII "Y".
```

Appendix A
Syntax Summary

This Appendix summarizes Basic-Debug Syntax in a meta-language descended from the Backus-Naur form. The language follows the rules below:

Syntactic constructs are denoted by lower case English words or phrases not enclosed in any special characters. Examples are command, stmnt, and gosub_stmnt.

The basic symbols of the language are keywords, written in upper case, and special characters, enclosed in quote marks. Examples are ',' LET '↑' NEW.

Possible repetition of a construct is indicated by appending either a '+', indicating one or more occurrences, or a '*', indicating zero or more occurrences. For example, the definition of number as digit+ means that a number consists of one or more digits.

Parentheses group together a number of constructs so that a repetition symbol (+ or *) may be applied to the group.

Square brackets denote optional items. The construct within the brackets may appear either zero or one times.

The vertical bar '|' signifies that one of several alternate constructs may be specified.

Curly brackets, '{' and '}', surround an English description of a construct that cannot be easily described otherwise.

```
command          => statement_line

program          => (number  statement_line)+

statement_line => (initial_stmnt ':')* stmnt {carriage return}

initial_stmnt    => go_stmnt
                 => in_stmnt
                 => input_stmnt
                 => let_stmnt
                 => print_stmnt

stmnt            => initial_stmnt
                 => if_stmnt
                 => gosub_stmnt
                 => goto_stmnt
                 => list_stmnt
                 => new_stmnt
                 => rem_stmnt
                 => return_stmnt
                 => run_stmnt
                 => stop_stmnt

go_stmnt         => GO '@' address [',' arg1 [',' arg2]]

address          => expression

arg1             => expression

arg2             => expression

gosub_stmnt      => GOSUB expression

goto_stmnt       => GOTO expression

if_stmnt         => IF expression relational_op expression [THEN] apodosis

relational_op    => '=' | '<>' | '>' | '<' | '>=' | '<='

apodosis         => number | statement_line

in_stmnt         => IN variable (',' variable)*

input_stmnt      => INPUT variable (',' variable)*

let_stmnt        => [LET] left_part '=' expression

left_part        => variable | '@' factor | '↑' factor

list_stmnt       => LIST [starting_line [',' ending_line]]

starting_line    => expression

ending_line      => expression

new_stmnt        => NEW

print_stmnt      => PRINT [item (delimiter item)*] [delimiter]
                 => initial_item (delimiter item)* [delimiter]

delimiter        => ',' | ';'

item             => quoted_string | expression | HEX '(' expression ')'
```

```
initial_item     => quoted_string | signed_expression | '(' expression ')'

quoted_string    => '"' {any character sequence, not containing nulls,
                         deletes, line feeds, carriage returns, escapes,
                         back spaces or quotes} '"'

rem_stmnt        => REM  {all following characters up to the end of line}

return_stmnt     => RETURN | RET

run_stmnt        => RUN [expression (',' expression)*]

stop_stmnt       => STOP

expression       => [add_op] term (add_op term)*

signed_expression
                 => add_op term (add_op term)*

add_op           => '+' | '-'

term             => factor (mult_op factor)*

mult_op          => '*' | '/' | '\'

factor           => variable
                 => '@' factor
                 => '↑' factor
                 => number
                 => '&' hex_number
                 => AND '(' expression [',' expression] ')'
                 => USR '(' address [',' arg1 [',' arg2 ]]')'
                 => '(' expression ')'

variable         => letter

number           => digit+

hex_number       => hex_digit*

letter           => A | B | C | D | E | F | G | H | I | J | K | L | M
                 => N | O | P | Q | R | S | T | U | V | W | X | Y | Z

digit            => 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

hex_digit        => digit | A | B | C | D | E | F
```

Appendix B

Baud Rate Switch Settings

The baud rate for the Z8671 system UART is set in timer T0 according to the least significant three bits of external memory location FFFD. Basic/Debug reads FFFD once at power-on/reset, and sets the UART according to the bit pattern. The bit patterns and corresponding baud rates are shown in the table below.

Content of Location FFFD			Baud Rate
		LSB	
1	1	1	300
1	1	0	110
1	0	1	1200
1	0	0	2400
0	1	1	4800
0	1	0	9600
0	0	1	19200
0	0	0	150

Appendix C
Error Code Summary

Basic/Debug gives numeric error messages. Error code meanings are given below. Error numbers not listed here occasionally appear. These unlisted codes mean either a memory overflow caused by a combination of too much program and too complicated an expression or too many GOSUBs, or just too complicated an expression, possibly coupled with too long a previous input line in a No-Ram system.

CODE	MEANING
0	Program interrupted by typing ESC code, or by Flag bit 0.
11	Program line has a line number 0 or greater than 32768.
17	Memory full; new line not inserted.
26	No program to RUN.
37	GOTO is not at the end of program line.
41	Cannot GOTO negative or zero line number.
44	Line number in GOTO does not exist.
66	GOSUB is not at the end of the line.
71	Unrecognizable statement type beginning with GO.
81	Unrecognizable statement type, or "=" missing from LET statement.
98	LET is missing its "=".
140	Quote missing in PRINT statement.
171	RETURN is not at the end of the line.
172	GOSUB stack underflow.
175	The GOSUB for this RETURN no longer exists.
181	STOP is not at the end of the line.

207	INPUT variable name is missing.
210	IN or INPUT expects variable name.
247	LIST is not at end of line.
310	Unrecognizable relation in IF statement.
346	Out of memory on GOSUB or expression evaluation.
381	Divide by zero.
391	Missing parenthesis in AND or USR call.
427	Syntax error in expression, or unrecognizable statement type.
431	Missing right parenthesis in expression.