ZBI Commands

This section explains the Zebra Basic Interpreter, its commands, descriptions, formats, and parameters.

Introduction to Zebra Basic Interpreter (ZBI)

What is ZBI and why is it for me?

ZBI is an "on-the-printer" programing language that offers many of the functions found in ANSI BASIC. The ZBI language allows the user to create applications that are run on the printer to manipulate data streams. By using ZBI, it is possible to have the printer perform the same functions that a computer or programmable terminal might otherwise be used for.

With the connectivity options available on Zebra printers, you may not need a separate computer. Simply load a ZBI program on your printers, add them to your network, and let the printers serve as the gateway for moving data.

Here are some of the applications that can be written using ZBI:

- Connect a barcode scanner to the printer. Based on scanned data, reprint tags, verify printed output, and manage a list of items.
- Connect a scale to the printer and print labels, tags, or receipts based on the weight of an item.
- Connect the printer to a PC-based database and send queries from the printer to retrieve or upload data.
- Convert incoming data into the commands that can be used to print a label. This is useful for replacing other brands of printers with new Zebra units.
- Provide fail-over to another printer when the target printer is in an error state.

Printers, ZBI Keys, & ZBI Versions

Information about ZBI 1.x and ZBI 2.x:

ZBI versions 1.0 through 1.5: ZBI 1.x was available on printers with X.10 or higher firmware (such as V48.10.x). To determine if the printer supports ZBI version 1, check the firmware version loaded on the printer. This can be determined by the absence of a “Z” in the firmware version number (for example, firmware V60.13.0.12 supports ZBI version 1, while V60.13.0.12Z does not).

ZBI-Developer can be used to create programs for use on printers that support ZBI version 1.x., however, the features that are only available in ZBI v2.x cannot be used with printers running ZBI v1.x. For example, “on-printer” debugging advanced file encryption and commands added in ZBI 2 are not supported in printers running ZBI 1.x. If you do not have a printer that meets this requirement, contact your reseller.

NOTE: Support for ZBI versions 1.0 through 1.5 is limited to syntax checking only. On-printer debugging is not supported for ZBI versions 1.0 through 1.5.

ZBI versions 2.0 and higher: Printers with firmware versions X.16 or later (for example, V60.16.x and V53.16.x) can support ZBI version 2.0 and later.

These printers can be either ZBI-Ready or ZBI-Enabled, depending on whether or not a ZBI Key file has been loaded on the printer. ZBI Keys can be loaded onto printers during manufacturing or later purchased at www.zebrasoftware.com. A Downloader Utility/ZBI Key Manager software utility is available to assist in the task of sending ZBI Keys to printers.

The ZBI.nrd file is required to be present on the printer for ZBI 2.0 to be enabled. The ZBI Key is stored on the printer's E: memory location with the name ZBI.nrd. The file is persistent. It cannot be deleted even if the printer's memory is initialized. For example, if the **^JB** command is used to initialize the location, the ZBI Key file will not be deleted.

When a printer is ZBI-Ready but not ZBI-Enabled, the firmware version will display a "Z" at the end of the version string (for example, V60.16.0Z). Additionally, the printer's configuration label will show that the printer is not ZBI-Enabled.

When a printer is ZBI-Enabled, the firmware version will not display a "Z" at the end of the version string (for example, V60.16.0Z). Additionally, the printer's configuration label will show that the printer is ZBI-Enabled.

NOTE: Each single ZBI Key can only be used once. When multiple printers are to be ZBI-Enabled, multiple Keys will be needed. The ZBI Key cannot be retrieved from printer to a host system.

Command and Function Reference Format

This section describes how commands and functions are presented in this document.

Command/Function NAME

Describes how the command is used, its capabilities, and its characteristics.

Format The Format section explains how the command is arranged and its parameters. For example, the **AUTONUM** command starts the auto-numbering option. The format for the command is **AUTONUM <A>,<B>**. The **<A>** and **<B>** are parameters of this command and are replaced with values determined by the user.

For functions, parameters are enclosed within parentheses and separated by commas, such as **EXTRACT$(A$,START$,STOP$).**

Numeric parameters are written as a name, while string parameters are written as a name followed by a dollar sign.

Parameters If a command has parameters that make a command or function more specific, they are listed under this heading. Still using the **AUTONUM** example, the **<A>** parameter is defined as:

<A> = number used to start the auto-numbering sequence

Return Value (functions only)

The return value is the result of evaluating the function or expression.

Example When a command is best clarified in a programming context, an example of the ZBI code is provided. Text indicating parameters, exact code to be entered, or data returned from the host is printed in the **Courier** font to be easily recognizable.

An example of **PRINT** code is:

```
10 PRINT "HELLO WORLD"
RUN
HELLO WORLD
```

Comments This section is reserved for notes that are of value to a programmer, warnings of potential command interactions, or command-specific information that should be taken into consideration. An example comment could be: This is a program command and must be preceded by a line number.

Function Rules

Functions built into this interpreter can be used in expressions only. The function names are not case sensitive.

If input parameters exist, they are enclosed in parentheses. If no parameters exist, no parentheses are used.

Variables referenced in the functions could be substituted by functions or expressions of the same type. If the function name ends with a $, it returns a string value. Otherwise, it returns a numeric value.

Section Organization

The sections in this guide are arranged based on programming topics. A brief description of the sections is listed below.

Editing Commands This section describes the commands which are used to manipulate the interpreter and enter programs.

Running and Debugging Outlines the control commands used to run and debug programs.

Base Types and Expressions Fundamental structure for manipulating strings and computing numeric and boolean values.

Control and Flow Commands to conditionally execute code and control the flow of the program

Input and Output Outlines how to communicate with the physical ports, internal ports, and network.

File System Shows how programs and formats can be saved and recalled

Comma Separated Values Identifies how to load and store comma separated data

Events Explains how to capture and trigger internal events in the printer

Systems Contains miscellaneous systems interface functions

String Functions Handles string manipulation

Math Functions Handles mathematical calculations

Array Functions Describes how to search, resize, and query arrays

Time and Date Functions Functions to access the real time clock option

Set/Get/Do Interface Functions to directly interface with the Set/Get/Do system

Example Programs More examples to give a head start in creating your applications

Writing ZBI Programs

There are two main ways to develop ZBI programs. The preferred method is to use the ZBI-Developer application. ZBI-Developer allows you to create and test programs before a printer is even turned on. In addition, many features of this program allow for quicker program creation and more meaningful debugging. ZBI-Developer can be downloaded from the Zebra web site.

An alternate method for developing a program is through a direct connection to the printer using a terminal emulation program.

Editing Commands

This section details the Editing Commands. This section describes the commands which are used to manipulate the interpreter and enter programs. These commands are used while controlling the ZBI environment from a console connection. Here is a quick list of these commands:

NEW – Clears out the program and variables currently in memory

REM and ! – Comment commands

LIST – Lists the program currently in memory

AUTONUM – Automatically generates the next line number

RENUM – Renumbers the program currently in memory

ECHO – Controls whether characters received on the console are echoed back

If you are using ZBI-Developer, the commands that will be most useful are AUTONUM and REM/!.

The following example shows the use of Editing commands from within a console connection.

Preview:

```
NEW
AUTONUM 10,5
10 REM "Hello World" Application
15 PRINT "Hello World" ! comment...
20

LIST
10 REM "Hello World" Application
15 PRINT "Hello World"

NEW
LIST
```

Entered automatically when AUTONUM is used

A blank line stops AUTONUM

Preview when viewed in ZBI-Developer

```
AUTONUM 10,5
REM "Hello World" Application
PRINT "Hello World" ! comment...
```

NEW

This command clears the interpreter's memory, including the line buffer and variables, but not any open ports. Use this command when creating code to restart the coding process or before resending a program from a file to the interpreter.

Format `NEW`

Parameters N/A

Example This is an example of how to use the `NEW` command:

```
10 PRINT "Hello World"
RUN
Hello World

LIST
10 PRINT "Hello World"

NEW
LIST
```

Comments This is an interactive command that takes effect as soon as it is received by the printer.

REM

A numbered **remark** line starts with **`REM`** and includes text in any form after it. This line is ignored by the interpreter.

Format `REM <comment>`

Parameters The comment string can contain any character and is terminated by a carriage return.

Example This is an example of how to use the `REM` command:

```
10 REM COMMAND LINES 20-100 PRINT A LABEL
```

Comments Remarks are used for program description and are included as a separate program line. To append a comment to the end of a program line, use the exclamation mark (`!`).

A useful method to keep comments in a stored file (but not in the printer) is to always start the REM line with the number 1. When all of the lines are sent to the printer, only the last REM line will stay resident in the printer. This will require less RAM for large programs.

Example This is an example of how to re-use the `REM` command:

```
1 REM MYPROGRAM COPYRIGHT ME Inc. 2008
1 REM While debugging a port may be left open
5 CLOSE ALL
1 REM Open the ports this program will use
10 OPEN #0: NAME: "SER" ! Restart the console
```

! (EXCLAMATION MARK)

The exclamation mark is the marker for adding comments to the end of numbered programming lines. Any text following the ! is ignored when the line or command is processed.

Format `!<comment>`

Parameters The comment string can contain any character and is terminated by the carriage return.

Example This is an example of how to use the `!` (comments) command:

```
10 LET A=10 ! Indicates number of labels to print
```

Comments None

LIST

This command lists the program lines currently in memory.

Format

LIST

LIST <A>

LIST <A>-<B>

Parameters

`default` = lists all lines in memory

<A> = line to start listing the program

<B> = line to stop listing the program. If not specified, only the line at **<A>** will print.

Example This is an example of how to use the `LIST` command:

```
1 REM MYPROGRAM COPYRIGHT ME Inc. 2008
1 REM While debugging a port may be left open
5 CLOSE ALL
1 rem Open the ports this program will use
10 OPEN #0: NAME: "SER" ! Restart the console
20 PRINT #0: "Hello World"
LIST
1 REM Open the ports this program will use
5 CLOSE ALL
10 OPEN #0: NAME: "SER" ! Restart the console
20 PRINT #0: "Hello World"

LIST 1
1 REM Open the ports this program will use

LIST 5-10
5 CLOSE ALL
10 OPEN #0: NAME: "SER" ! Restart the console
```

Comments The output of the **LIST** command may not match exactly what was entered. It is based on how the program lines are stored in memory. Notice that the last comment line the REM is entered in lower case characters. When it is listed, the REM is displayed in uppercase.

This is an interactive command that takes effect as soon as it is received by the printer.

AUTONUM

This command automatically generates sequential program line numbers.

Format `AUTONUM <A>,<B>`

Parameters

A = the number used to start the auto-numbering sequence

B = the automatic increment between the new line numbers

Example This example shows specifying the starting line number in the increment between new line number. Type the following at the prompt:

```
AUTONUM 10,5
SUB START
PRINT "HELLO WORLD"
GOTO START

LIST
```

Will produce:

```
AUTONUM 10,5
10 SUB START
15 PRINT "HELLO WORLD"
20 GOTO START
```

The three lines are automatically started with the **AUTONUM** parameters; in this case, the first line starts with 10 and each subsequent line increments by 5.

Comments This feature is disabled by overwriting the current line number and entering the desired interactive mode commands, or leaving the line blank.

Use of the **SUB** command allows for **GOTO** and **GOSUB** statements that do not require line numbers in your program.

This is an interactive command that takes effect as soon as it is received by the printer.

RENUM

This command renumbers the lines of the program being edited. **RENUM** can reorganize code when line numbers become over- or under-spaced. The line references following **GOTO** and **GOSUB** statements are renumbered if they are constant numeric values. Renumbering does not occur if the line numbers are outside of the range limits of 1 to 10000.

Format **RENUM <A>,<B>**

Parameters

<A> = the number to start the renumbering sequence

<B> = the automatic increment between the new line numbers

Example This is an example of how to use the RENUM command:

```
LIST
13 LET A=6
15 LET B=10
17 GOTO 13
RENUM 10,5
LIST
10 LET A=6
15 LET B=10
20 GOTO 10
```

NOTE: The target of the GOTO command changes from 13 to 10 to reflect the renumbering.

Comments This is an interactive command that takes effect as soon as it is received by the printer.

ECHO

When Console Mode is enabled, this command controls whether the printer echoes the characters back to the communications port. If **ECHO ON** is entered, keystroke results return to the screen. If **ECHO OFF** is entered, keystroke results do not return to the screen.

Format

ECHO ON

ECHO OFF

Parameters

<ON/OFF> = toggles the ECHO command on or off

Example N/A

Comments This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

Running and Debugging Commands

The following commands were written before the development of the ZBI-Developer application. With that application, and when using ZBI version 1, the following commands are essentially obsolete. However, for those who started developing ZBI applications before ZBI-Developer, the following reference will be helpful.

RUN – Starts executing the program currently in memory at the first line of the program

CTRL-C Sends an end-of-transmission character, **ETX** , to the console to terminate the ZBI program currently running.

RESTART – Starts executing the program currently in memory where it was last stopped

STEP – Executes one line of the program in memory where it was last stopped

DEBUG – This mode controls whether or not the TRACE and BREAK commands are processed

TRACE – Shows which lines have been executed and which variables have been changed

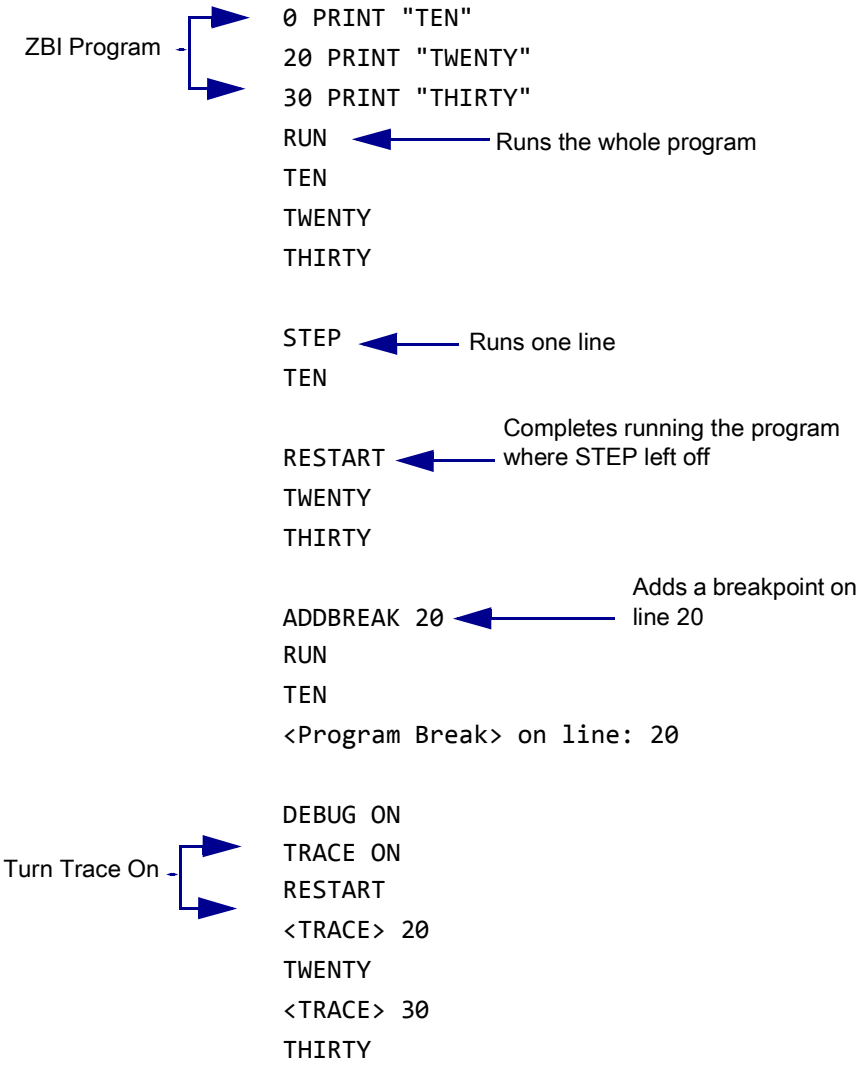
BREAK – Stops the currently running program

ADDBREAK – Adds a break to an existing line

DELBREAK – Deletes an existing break

ZPL Terminates and exits the ZBI environment.

Example This example shows many of the Running and Debug Commands in practice.



RUN

This command executes the current program, starting with the lowest line number. The interpreter will continue to execute the program lines in order unless a control statement directs the flow to a different point. When a higher line number does not exist or the END command is processed, the **RUN** command will stop.

Format **RUN**

Parameters N/A

Example This is an example of how to use the RUN command:

```
10 PRINT "ZBI"
20 PRINT "Programming"
RUN
ZBI
Programming

15 END
RUN
ZBI
```

Comments Ports that are open when the application is activated will remain open after the application has terminated. Variables also remain after the application has terminated.

To execute programs when the printer is powered on, use the **^JI** command in the Autoexec.zpl file.

This is an interactive command that takes effect as soon as it is received by the printer.

CTRL-C

Sending an end-of-transmission character, **ETX** (3 in hex), to the console (port 0) terminates the ZBI program currently running.

Format N/A

Parameters N/A

Example N/A

Comments In most terminal programs, you terminate the program using the **Ctrl-C** key sequence. Another method is to store an ETX character in a file and have the terminal program send the file to the console port.

NOTE: It is not recommended to use **RESTART** after using a **CTRL-C** because a command may have been prematurely interrupted. Restarting will have an undefined result.

RESTART

If a program was halted by a break point or the **BREAK** command, the **RESTART** command can be used to reactivate the program at the point it stopped. **RESTART** functions similar to **RUN**, except the program attempts to restart from the point where it was last terminated. It also works in conjunction with the **STEP** command, picking up where the **STEP** command ended.

Format **RESTART**

Parameters N/A

Example An example of the **RESTART** command:

```
10 PRINT "TEN"
20 PRINT "TWENTY"
30 PRINT "THIRTY"
RUN
TEN
TWENTY
THIRTY

STEP
TEN

RESTART
TWENTY
THIRTY

ADDBREAK 20
RUN
TEN
<Program Break> on line: 20

DEBUG ON
TRACE ON
RESTART
<TRACE> 20
TWENTY
<TRACE> 30
THIRTY
```

Comments If the program has not been run or has finished, **RESTART** runs the program from the beginning.

This is an interactive command that takes effect as soon as it is received by the printer.

STEP

If a program was stopped by a **BREAK** command, **STEP** attempts to execute the program one line from where it last ended. If the program has not been run or has been completed, this executes the lowest numbered line.

Format **STEP**

Parameters N/A

Example This is an example of how to use the **STEP** command:

```
10 PRINT "Hello World"
20 Print "TWENTY"
STEP
Hello World

STEP
TWENTY
```

Comments This is an interactive command that takes effect as soon as it is received by the printer.

DEBUG

DEBUG enables and disables the **TRACE** and **BREAK** commands.

Format

DEBUG ON

DEBUG OFF

Parameters

ON = turns the debug mode on enabling the **TRACE** and **BREAK** commands to be processed.

OFF = turns the debug mode off. This disables the **TRACE** mode and causes **BREAK** commands to be ignored.

Example See TRACE on page 452 and BREAK on page 453.

Comments This command has no effect on the processing of break points in ZBI-Developer. It is recommended that you avoid using the **DEBUG** command when writing programs in the ZBI-Developer environment, instead use the Debug capabilities of ZBI-Developer.

TRACE

This command enables you to debug an application by outputting the executed line numbers and changed variables to the console.

Format

TRACE ON

TRACE OFF

Parameters

<ON/OFF> = controls whether **TRACE** is active (ON) or disabled (OFF).

If **DEBUG** is activated and the **TRACE** command is on, trace details are displayed. When any variables are changed, the new value displays as follows:

<TRACE> Variable = New Value

Every line processed has its line number printed as follows:

<TRACE> Line Number

Example An example of TRACE command in use:

```
10 LET A=5
20 GOTO 40
30 PRINT "Error"
40 PRINT A
DEBUG ON
TRACE ON
RUN
<TRACE> 10
<TRACE> A=5
<TRACE> 20
<TRACE> 40
5
```

Comments This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

It is recommended that you avoid using the **TRACE** command when writing programs in the ZBI-Developer environment, instead use the Debug capabilities of ZBI-Developer.

BREAK

This command allows you to stop the program when the program reaches this line.

Format **BREAK**

Parameters N/A

Example An example of BREAK command in use:

```
10 LET A=5
20 BREAK
30 PRINT A
DEBUG ON
TRACE ON
RUN
<TRACE> 10
<TRACE> A=5
<TRACE> 20
<USER BREAK>
```

Comments This command is available only when the **DEBUG** function has been activated. When **DEBUG** is on, **BREAK** halts processing. **RUN** starts the program from the beginning. **RESTART** allows the program to continue from where it left off.

When using ZBI-Developer, this command will interfere with the debugging operations built into the application.

This is a program command that must be preceded by a line number.

ADDBREAK

Description This command allows you to stop the program when the program reaches a specified line.

Format `ADDBREAK <A>`

Parameters

A = the line number to break on. If the number specified is not in the program, the program will not break.

Example An example of the ADDBREAK command.

```
10 LET A=5
20 PRINT A
ADDBREAK 20
RUN
<PROGRAM BREAK> ON LINE:20

RESTART
5
```

Comments This command is available only when the DEBUG function has been activated. When DEBUG is on, BREAK halts processing. RUN starts the program from the beginning. RESTART allows the program to continue from where it left off.

This is the command used internally by ZBI-Developer when the user right-clicks over a program line and adds a Breakpoint via the "Toggle Breakpoint" selection.

It is the recommended method for setting breakpoints in ZBI.

A maximum of 16 breakpoints can be set in an application.

This is an interactive command that takes effect as soon as it is received by the printer.

DELBREAK

This command allows you to remove existing breakpoints.

Format `DELBREAK <A>`

Parameters `A` = the line number from which to remove the break. If 0 is specified, all break points will be removed. If the number specified is not a breakpoint, the command will have no effect.

Example An example of the DELBREAK command:

```
10 LET A=5
20 PRINT A
ADDBREAK 20
DEBUG ON
TRACE ON
RUN
<TRACE> 10
<TRACE> A=5
<PROGRAM BREAK> ON LINE:20

RESTART
<TRACE> 20
5

DELBREAK 20
RUN
<TRACE> 10
<TRACE> A=5
<TRACE> 20
5
```

Comments This command is available only when the `DEBUG` function has been activated. When `DEBUG` is on, `BREAK` halts processing, `RUN` starts the program from the beginning, and `RESTART` allows the program to continue where it left off.

This is the command used internally by ZBI-Developer when the user right-clicks over a program line and removes a Breakpoint via the "Toggle Breakpoint" selection.

A maximum of 16 breakpoints can be set in an application.

This is an interactive command that takes effect as soon as it is received by the printer.

ZPL

This command terminates and exits the ZBI environment.

Format **ZPL**

Parameters N/A

Example An example of the ZPL command.

```
ZPL
ZBI TERMINATED
```

Comments This is an interactive command that takes effect as soon as it is received by the printer.

Base Types and Expressions

There are two base types in the ZBI language. These types are Integers and Strings. Integers are whole numbers that contain no fractional part. The range of values for integers is:

-2,147,483,648 to +2,147,483,647

Strings are character arrays. The string length is only limited by the amount of memory in the system (version 2.0 and higher). Each character can have a value between 0 and 255 (version 2.0 and higher).

The use of control characters (0-31) may be difficult to debug based on the handling of control characters in different communications programs. In addition the ETX (3) will terminate a ZBI application when it is received on the console port. Use the CHR$ function when control characters must be placed into strings.

NOTE: In ZBI version 1.4 and lower, there was a string length limit of 255 characters.

This section is organized as follows:

- Variable Names
- Variable Declarations
- Constants
- Arrays
- Assignment
- Numeric Expressions
- String Concatenation (&)
- Sub-strings
- Boolean Expressions
- Combined Boolean Expressions

Variable Names

To distinguish strings from integers, string variable names must end in a $. Variable names must start with a letter and can include any sequence of letters, digits, and underscores. Function and command names must not be used as a variable name. Variable names are not case sensitive and are converted to uppercase by the interpreter.

A common mistake is to use a command or function name as a variable. To avoid using these reserved words, ZBI-Developer can be a useful resource. Reserved words are highlighted making it easier to spot this occurrence and thus, saving debugging time.

Valid variable names:

I, J, K, VARNAME, VARSTR$, MYSTR$,MY_STR9$

Invalid Names:

STR$ = Reserved word

ORD = Reserved word

VAL = Reserved word

W# = Invalid character (#)

9THSTR = Variable can not start with a number

Variable Declarations

ZBI will allow storage of up to 255 variables. If more variables are needed, consider using arrays to store data. The base array will take up one of the 255 variable slots, but it can be declared to allow for many indices.

Variables can be declared explicitly or implicitly. If a variable has not been used before, it will be declared when used. The default value for an integer will be zero and the default value of a string will be an empty string.

Explicit:

```
DECLARE NUMERIC <variable_name>
DECLARE STRING  <variable_name$>
```

If the variable existed before the DECLARE statement, it will be defaulted.

Implicit:

```
LET <variable_name> = NUMERIC EXPRESSION
LET <variable_name$> = STRING EXPRESSION
```

The Interpreter is limited to 255 variables. If more variables are required, consider using arrays.

Constants

Integers are represented simply by numbers, such as 5, -10, 10000. Do not use commas in integer constants. Strings are enclosed by quotes. If a quote is required in the string, use double quotes, such as “Look here->””<-“ would result in the string – Look here->”<-.

Arrays

An array is a collection of string or integer values used by a program. Array indices are accessed through parentheses. Array indices start at 1 and end at the length of an array (for example, MyArray(3) returns the value in the third location of the variable array). One- and two-dimensional arrays are allowed. Two-dimensional arrays are referenced with two indices in parentheses, separated by a comma.

Arrays must be allocated through the use of the **DECLARE** command. Arrays can be re-dimensioned by using **DECLARE**, however, this will replace the original array.

Array size is limited only by the size of the memory available.

Format

DECLARE STRING <ARRAYNAME$>(<SIZE>)

DECLARE STRING <ARRAYNAME$>(<ROWS>,<COLUMNS>)

DECLARE NUMERIC <ARRAYNAME>(<SIZE>)

DECLARE NUMERIC <ARRAYNAME>(<ROWS>,<COLUMNS>)

Parameters

<SIZE> = number of entries in a single dimension array

<ROWS> = number of rows in a two dimensional array

<COLUMNS> = number of columns in a two dimensional array

Example An example of **ARRAY** code is:

```
10 DECLARE STRING INARRAY$(3)
20 FOR I = 1 TO 3
30 PRINT "Name "; I; ": ";
40 INPUT INARRAY$(I)
50 NEXT I
60 PRINT INARRAY$(1); ", "; INARRAY$(2); ", and "; INARRAY$(3);
70 PRINT " went to the park"
RUN
Name 1: Jim
Name 2: Jose
Name 3: Jack
Jim, Jose, and Jack went to the park
```

Comments If you attempt to access an array outside of its allocated bounds, an error will occur.

Assignment

All lines must start with a command. In order to assign a value to a variable, use the LET command. Multiple variables can be placed before the =. The variable types must match the expression type.

The right side of the assignment is always calculated completely before the assignment is made. This allows a variable to be the target and source of the assignment.

When a value is assigned to a string variable with a sub-string qualifier, it replaces the value of the sub-string qualifier. The length of the value of the string variable may change as a result of this replacement.

Example An **ASSIGNMENT** example:

```
10 LET A=5
20 LET B$="HELLO"
30 LET B$(5:5)=B$
```

LET

The LET command is used to assign value to a specific variable. The expression is evaluated and assigned to each variable in the variable list. See Assignment on page 460.

Format

LET <variable> [,<variable>]* = <expression>

The variable types must match the expression type or an error message will be displayed.

Error: Poorly formed expression.

When a value is assigned to a string variable with a sub-string qualifier, it replaces the value of the sub-string qualifier. The length of the value of the string variable may change as a result of this replacement.

Parameters N/A

Example This is an example of how to use the **LET** command:

```
10 LET A$= "1234"
15 LET A$(2:3)= "55" ! A$ NOW = 1554
20 LET A$(2:3)= "" ! A$ NOW = 14

10 LET A$= "1234"
15 LET A$(2:3)= A$(1:2) ! A$ NOW = 1124

10 LET A$= "1234"
15 LET A$(2:1)= "5" ! A$ NOW = 15234
```

Comments This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

Numeric Expressions

A base numerical expression can be either a constant, variable, or another numerical expression enclosed in parentheses. The five types used (addition, subtraction, multiplication, division, and exponentiation) are listed below. When evaluating an expression exceeding the maximum or minimum values at any point creates an undefined result. (maximum value: 2,147,487,647; minimum value: -2,147,483,648)

Floating point is not supported.

When using division, the number is always rounded down. For example, 5/2=2. Use **MOD** to determine the remainder.

Format

1. + (addition) Addition expressions use this format:

   <A>+<B>

   5+2 result = 7

   VAL ("25") +2 result =27

2. – (subtraction) Subtraction expressions use this format:

   <A>-<B>

   5-2 result = 3

   VAL ("25") -2 result =23

3. * (multiplication) Multiplication expressions use this format:

   <A>*<B>

   5*2 result = 10

   VAL ("25") *2 result =50

4. / (division) Division expressions use this format:

   <A>/<B>

   5/2 result = 2

   VAL ("25") /2 result =12

5. ^ (exponentiation) Exponentiation expressions use this format:

   <A>^<B>

   5^2 result = 25

   VAL ("25") ^2 result =625

Order of Precedence

In mathematics, the order of precedence describes the sequence that items in an expression are processed. All expressions have a predefined order of precedence.

The order of precedence is listed below:

Functions

Parenthetical Expressions ()

^

* and /

+ and -

The * and / have the same precedence, and the + and - have the same precedence. Items with the same order of precedence are processed from left to right.

For example, this expression 5+(8+2)/5 is processed as 8+2=10, followed by 10/5=2, then 5+2 to give a result of 7.

Functions and parenthetical expressions always have the highest order of precedence, meaning that they are processed first.

String Concatenation (&)

The basic string expression may be either a constant or a variable, and concatenation (&) is supported. Using the concatenation operator (&) adds the second string to the first string.

```
<A$> & <B$>
```

Example This is an example of how to use the `STRING CONCATENATION (&)` command:

```
10 LET A$= "ZBI-"
20 LET B$= "Programming"
30 LET C$= A$ & B$
40 PRINT C$
RUN
ZBI-Programming
```

Sub-strings

Using a sub-string operator on a string allows a specific portion of the string to be accessed. This portion may be the target of an assignment operation or a reference to a portion of the string. To determine the coordinates of the string portion to be used, count the characters from the beginning to the end of the string, including spaces.

Format

LET <STRVAR$>(<A>:<B>)=<C$>

LET <C$> = <STRVAR$>(<A>:<B>)

Parameters

<A> = the position of the first character in the desired string

<B> = the position of the last character in the desired string.

<STRVAR$> = base string variable

If the A parameter is less than 1, it is automatically assigned a value of 1. Because the string is calculated starting with 1, the A parameter cannot be less than 1.

If B is greater than the length of the string, it is replaced with the length of the string.

If A is greater than B, a NULL string (""), which points to the location of the smaller of A or the end of the string, is returned. This is used when adding a string in the middle of another string without removing a character.

Example This is an example of a sub-string reference:

```
LET A$="Zebra Quality Printers"
LET B$=A$(1:13)
PRINT B$
Zebra Quality

This is an example of a sub-string assignment.
LET A$= "1234"
LET A$(2:3)= "55" ! A$ NOW = 1554
LET A$(2:3)= "" ! A$ NOW = 14

LET A$= "1234"
LET A$(2:3)= A$(1:2) ! A$ NOW = 1124

LET A$= "1234"
LET A$(2:1)= "5" ! A$ NOW = 15234
```

The best way to think of assignment to a sub-string is as follows: an assignment is like selecting a word, and pasting over the selection with the new string.

Boolean Expressions

A Boolean expression holds 0 (zero) as false and non-zero as true.

Formats

```
<STRING EXPRESSION> <BOOLEAN COMPARE> <STRING EXPRESSION>
<NUMERIC EXPRESSION> <BOOLEAN COMPARE> <NUMERIC EXPRESSION>
NOT(<BOOLEAN EXPRESSION>)
```

Parameters

<STRING EXPRESSION> = a string variable, string constant or any combination with concatenation

<NUMERIC EXPRESSION> = any mathematical operation

Comments A numeric expression cannot be compared to a string expression.

Numeric expressions can substitute a Boolean expression where a value of 0 (zero) represents false and a non-zero value represents true.

Base Boolean expressions:

1. < (less than)

Expression	Result
1< 2	true
2<2	false
2<1	false

2. <= (less than or equal to)

Expression	Result
1<=2	true
2<=2	true
2<=1	false

3. > (greater than)

Expression	Result
1> 2	false
2>2	false
2>1	true

4. >= (greater than or equal to)

Expression	Result
1>=2	false
2>=2	true
2>=1	true

5. = (equal to)

Expression	Result
1=2	false
2=2	true
"A"="AA"	false
"A"="A"	true

6. <> (not equal to)

Expression	Result
1<>2	true
2<>2	false
"A"<>"AA"	true
"A"<>"A"	false

Combined Boolean Expressions

AND, **OR**, and **NOT** can be used in conjunction with base Boolean expressions to recreate expanded Boolean expressions.

1. **NOT** — Negate the target expression.

Expression	Result
NOT 1=2	true
NOT 1=1	false

2. **AND** — Both expressions must be true for a true result.

Expression	Result
1=2 AND 1=2	false
2=2 AND 1=2	false
1=2 AND 2=2	false
2=2 AND 2=2	true

3. OR — If either expression is true, the result will be true.

Expression	Result
1=2 OR 1=2	false
1=2 OR 2=2	true
2=2 OR 1=2	true
2=2 OR 2=2	true

Order of Precedence

The order of precedence is listed below:

Expressions and Functions

Parenthetical expressions ()

<, <=, <>, =, =>, >

NOT, AND, OR

Control and Flow

This section outlines the commands to conditionally execute code and control the flow of the program. Here is a quick list of these commands:

IF Statements Executes or skips a sequence of statements, depending on the value of a Boolean expression.

DO Loops Repeats instructions based on the results of a comparison.

FOR Loops A control flow statement which allows code to be executed iteratively.

GOTO/GOSUB Causes an unconditional jump or transfer of control from one point in a program to another.

SUB Allows you to “substitute” names instead of actual line numbers as the target of **GOSUB**s and **GOTO**s.

EXIT Used to exit the **DO** and **FOR** loops.

END Terminates any program currently running.

IF Statements

If the value of the **<Boolean expression>** in an **IF** statement is true and a program line follows the keyword **THEN**, this program line is executed. If the value of the Boolean expression is false and a program line follows the keyword **ELSE**, this program line is executed. If **ELSE** is not present, then execution continues in sequence, with the line following the **END IF** statement.

Nesting of blocks is permitted, subject to the same nesting constraints as **DO-LOOP**s (no overlapping blocks).

ELSE IF statements are treated as an **ELSE** line followed by an **IF** line, with the exception that the **ELSE IF** shares the **END IF** line of the original **IF** statement.

Format

```
IF <Boolean expression> THEN
~~BODY~~
[ELSE IF <Boolean expression> THEN
~~BODY~~]*
[ELSE
~~BODY~~]
END IF
```

Parameters N/A

Example This is an example of how to use the `IF` statement command:

```
10 IF A$="0" TyHEN
20 PRINT "ZBI IS FUN"
30 ELSE IF A$="1" THEN
40 PRINT "ZBI IS EASY"
50 ELSE IF TIME=1 THEN
60 PRINT "It is one second past midnight"
70 ELSE
80 PRINT "X=0"
90 END IF
```

DO Loops

Processing of the loop is controlled by a **<WHILE/UNTIL>** expression located on the **DO** or **LOOP** line.

Processing a **WHILE** statement is the same on either the **DO** or **LOOP** lines. The Boolean expression is evaluated and if the statement is true, the **LOOP** continues at the line after the **DO** statement. Otherwise, the line after the corresponding **LOOP** is the next line to be processed.

Processing an **UNTIL** statement is the same on either the **DO** or **LOOP** lines. The Boolean expression is evaluated and if the statement is false, the **LOOP** continues at the line after the **DO** statement. Otherwise, the line after the corresponding **LOOP** is the next to be processed.

If **<WHILE/UNTIL>** is on the **LOOP** line, the **BODY** of the loop is executed before the Boolean expression is evaluated.

If neither the **DO** or **LOOP** line has a **<WHILE/UNTIL>** statement, the loop continues indefinitely.

Some notes about **DO-LOOP**s:

- can be nested
- cannot overlap
- have two formats

Format

```
DO [<WHILE/UNTIL> <Boolean expression>]
~~BODY~~
LOOP [<WHILE/UNTIL> <Boolean expression>]
```

Example This is an example of how to use the **DO-LOOP** command with the conditional on the DO line:

```
10 DO WHILE A$="70"
20 INPUT A$
30 LOOP
```

Example This is an example of how to use the **DO UNTIL LOOP** command with conditional on the LOOP line:

```
10 DO
20 INPUT A$
30 LOOP UNTIL A$="EXIT"
```

Comments This is a program command that is preceded by a line number.

FOR Loops

FOR loops are an easy way to iterate through a range of values and run a body of code for each value iterated.

Format

```
FOR <I> = <A> TO <B> [STEP <C>]
~~BODY~~
NEXT <I>
```

Parameters

<I> = indicates a numeric variable is used. <I> increments each time through the **FOR-LOOP**.

<A> = the value assigned to <I> the first time through the loop

<B> = the last value through the loop

<C> = (Optional) the amount <I> increments each time through the loop

Values of **I** for the following situations:

Statement	Result
FOR I=1 TO 10	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
FOR I=10 TO 1	{10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
FOR I=1 TO 10 STEP 2	{1, 3, 5, 7, 9}
FOR I=10 TO 1 STEP 2	{10, 8, 6, 4, 2}
FOR I=10 TO 1 STEP 2	{ } FOR LOOP skipped

Example This is an example of how to use the **FOR LOOP** command:

```
10 FOR X=1 TO 10 STEP 1
20 PRINT X; ":ZBI IS FUN"
30 NEXT X
```

Comments **FOR** loops can be nested but cannot overlap. Variables cannot be reused by the nested loops.

GOTO/GOSUB

GOSUB is followed by a line number. The program will attempt to process the line the **GOSUB** command points to rather than the next line of the program. Upon executing the **GOSUB** statement, the interpreter continues running at the line number specified following **GOSUB**. If the line number referenced does not exist, an error will occur.

Before executing the next line, the **GOSUB** command stores the line number of the **GOSUB** line. When the **RETURN** statement is called, the program moves back to the next line following the **GOSUB**.

Executing a **RETURN** statement without a corresponding **GOSUB** statement causes an error.

GOSUB statements can be nested.

GOTO works the same way as **GOSUB** except that no return address will be stored.

Format

GOSUB <A>

RETURN

GOTO <A>

Parameters **<A>** = the program location executed immediately after the `GOTO` or `GOSUB`.

Example This is an example of how to use the **GOSUB** command:

```
10 PRINT "Call Subroutine"
20 GOSUB 1000
30 PRINT "Returned from Subroutine"
40 END
1000 PRINT "In Subroutine"
1010 RETURN
```

Example This is an example of how to use the **GOTO** command:

```
10 PRINT "Prepare to Jump!"
20 GOTO 1000
30 PRINT "Jump Missed..."
1000 PRINT "Jump Successful"
1010 END
```

Comments These are program commands and must be preceded by line numbers.

SUB

This command allows you to use names instead of actual line numbers as the target of **GOSUB**s and **GOTO**s. **AUTONUM** can be used at the beginning of a file and there is no need to compute the line number where the jump will go.

Format **10 SUB <A>**

Parameters **<A>** = the integer variable to use as a target for the **GOTO**/**GOSUB**

Example This is an example of how to use the **SUB** command:

```
AUTONUM 1,1
GOSUB INITCOMM
DO
GOSUB GETINPUT
GOSUB PROCESSINPUT
LOOP
SUB INITCOMM
OPEN #1:NAME "SER"
RETURN
SUB GETINPUT
INPUT #1: A$
RETURN
SUB PROCESSINPUT
PRINT A$
RETURN
```

Comments **<A>** is a numeric variable. If this variable is changed in the program, any **GOSUB/GOTO** to this variable may fail.

EXIT

This command is used to exit the **DO** and **FOR** loops.

Format

EXIT DO

EXIT FOR

Parameters The specified loop type is exited. For the **DO** command, the program will continue execution on the line following the next **LOOP**. Likewise for the **FOR** command, the program will continue on the line after the next **NEXT** command.

Example N/A

Comments This is a program command that is preceded by a line number. To be explicit and reduce errors, it is recommended to use **GOTO** instead of **EXIT**.

END

The **END** command terminates any program currently running. When the **END** command is received, the interpreter returns to interpreting commands (**>**).

Format **END**

Parameters N/A

Example This is an example of how to use the **END** command:

```
10 PRINT "THIS PROGRAM WILL TERMINATE"
20 PRINT "WHEN THE END COMMAND IS RECEIVED"
30 END
40 PRINT "THIS SHOULD NOT PRINT"
RUN
THIS PROGRAM WILL TERMINATE
WHEN THE END COMMAND IS RECEIVED
```

Comments This is a program command and is preceded by a line number.

Input and Output

This section oulines how to communicate with physical ports, internal ports, and the network.

ZBI allows access to the physical and network connections in the printer. Most ports are, by default, connected to the ZPL processor. When a port is opened in ZBI, the port will be disconnected from ZPL and connected into the interpreter. Depending on the type of connection, there are two methods you may use to start the connection. For the static connections, the **OPEN** command should be used. These are the connections that you open when starting your program and leave open for the duration of your program. For dynamic connections, servers and clients are set up following the "Sockets" model. On servers, the actual connections are started upon successful calls to **ACCEPT**. Below are the available connections that can be made and the preferred accessors.

Available Ports

Port/Connection	ZBI Name	Preferred Access Commands/Functions
Serial	"SER"	OPEN, CLOSE
Parallel	"PAR"	OPEN, CLOSE
USB	"USB"	OPEN, CLOSE
ZPL parser	"ZPL"	OPEN, CLOSE
TCP Server	"TCP", "TCPX"	SERVERSOCKET, SERVERCLOSE, ACCEPT, CLOSE
TCP Client	"TCP"	CLIENTSOCKET, CLOSE
UDP Server	"UDP"	SERVERSOCKET, SERVERCLOSE, ACCEPT, CLOSE
UDP Client	"UDP"	CLIENTSOCKET, CLOSE
Email Sender	"EML"	OPEN, CLOSE
Bluetooth	"BLU"	OPEN, CLOSE
Note: TCPx will not work on PS2 or PS100 print servers.		

Creating Connections

Here is a quick list of the commands in this section:

OPEN Opens a port for transmitting and receiving data.

CLOSE Closes specific ports that are in use.

DATAREADY Determines if there is data received on a specified port.

SERVERSOCKET Opens a listening socket for incoming UDP packets or TCP connections.

SERVERCLOSE Closes a listening server socket.

CLIENTSOCKET Creates an outgoing TCP connection or sets up UDP transmissions.

ACCEPT Accepts incoming TCP or UDP connections and assigns a channel for the connection.

OPEN

This command is used to open a port for transmitting and receiving data.

Format `OPEN #<CHANNEL>: NAME <PORT$>`

Parameters

`<CHANNEL>` = a number to use as a handle to the port for all future communications

Values: 0 to 9

Default: a port must be specified

`<PORT$>` = port name to open. See Available Ports on page 473.

Example This is an example of how to use the `OPEN` command:

```
10 OPEN #1: NAME "ZPL"
```

The port being opened no longer allows data to pass directly into its buffer, it disconnects, and the interpreter now controls the data flow.

Data already in the buffer stays in the buffer.

Comments This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

CLOSE

This command is implemented to close specific ports that are in use. If a port is open on a channel and the `CLOSE` command is entered, the port closes and returns to communicating with the ZPL buffer.

Format

`CLOSE #<A>`

`CLOSE ALL`

Parameters

`<A>` = Numeric value of port to close

Values: 0 through 9

`All` = closes all open ports and network connections

NOTE: `CLOSE ALL` will close the console.

Example This example shows the closing of channel 1:

```
10 CLOSE #1
```

Comments This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

DATAREADY

This function is used to determine if there is data received on a specified port.

Format `DATAREADY (A)`

Parameters A = the port to check

Returns 1 if there is data, 0 if there is no data.

Example This is an example of how to check if there is a data on a port:

```
10 PRINT DATAREADY(0)
RUN
```

The result, assuming no data is waiting, is:

```
0
```

Comments If this command follows the **INPUT** command, it may return 1 if the line received was ended with a CRLF. In this case, **INBYTE** can be used to take the LF out of the buffer.

SERVERSOCKET

This function opens a listening socket for incoming UDP packets or TCP connections. It must be used in conjunction with the **ACCEPT** function.

Format `SERVERSOCKET (TYPE$,PORT)`

Parameters

TYPE$ = listens for any of the following communication protocols:

"TCP" = TCP – PORT parameter is ignored. The current port will be used.

"TCPX" = TCP – any open port

"UDP" = UDP – any open port

Returns

NUMERIC = returns the handle of the server upon success.

Example See the examples for TCP Server on page 489 and UDP Server on page 491.

Comments When using **TCPX**, care needs to be taken not to use a port that is already open on the printer. No error message will be returned until the **ACCEPT** function is called.

SERVERCLOSE

This function closes a listening server socket created by SERVERSOCKET.

Format `SERVERCLOSE(SOCKET)`

Parameters

`SOCKET` = the socket handle returned from a successful SERVERSOCKET invocation.

Returns Returns a 0 if the socket was already closed or a 1 if the socket was closed successfully.

Example This example shows how to close a listening server socket.

```
10 LET SERVER_HANDLE = SERVERSOCKET("TCPX", 19100)
20 LET SCERR = SERVERCLOSE(SERVER_HANDLE)
```

CLIENTSOCKET

This function creates an outgoing TCP connection or sets up UDP transmissions. Once set up for UDP, packets can be sent by printing to the socket. Packets are sent when the size limit is met or a `EOT` character is written.

Format `CLIENTSOCKET (TYPE$, IPADDR$, PORT)`

Parameters

`TYPE$` = set to "UDP" or "TCP".

`IPADDR$` = connects to this address.

`PORT` = connects to this IP port.

Returns The port number assigned to the connection.

Example See the examples for TCP Server on page 489 and UDP Server on page 491.

Comments Multiple communications connections can be made up to the maximum of 10. Each protocol may have a different limit based on the support of the print server used. Test the worst case situation based on your application's needs or use `ONERROR` to recover from failed connection attempts.

ACCEPT

This function will accept incoming TCP or UDP connections and assign a channel for the connection. **SERVERSOCKET** must be used to set up the listening socket before **ACCEPT** can be used.

Format **ACCEPT (SERVER, CLIENT_INFO$)**

Parameters

SERVER = the handle returned by the **SERVERSOCKET** call.

CLIENT_INFO$ = string variable will have the connecting client's IP address and port separated by a space when using UDP.

Returns The channel number to use to communicate with the client.

Example See the examples for TCP Server on page 489 and UDP Server on page 491.

Comments It is best to poll this function at regular intervals. When there is no connection waiting, this function will trigger an error. Follow this function with the **ON ERROR** command to divert to a section of code that handles an unsuccessful connection.

ACCEPT can be called before closing a previous connection. This allows for processing multiple incoming streams of data. There are limits on the number of simultaneous incoming connections based on the print server model on the printer.

Connection closure can be detected when any input or output command to the port triggers an error. These commands should be followed by an **ON ERROR** statement to send the program into a recovery state and to shutdown the connection cleanly.

Reading and Writing

This manual has detailed various functions to read and write to all of the ports. The following section gives an overview of the commands, functions, and when each should be used.

To start, it is important to understand the term "blocking". In communications code, a function or command is "blocking" if it waits for all of the requested data to be received before it returns.

INPUT (blocking) Reads one line into each string specified.

PRINT (blocking) Simple method to write specified expressions out.

OUTBYTE (blocking) Writes one byte out.

INBYTE (blocking) Reads in one byte.

READ (non-blocking) Reads in all available data up to the maximum amount specified.

WRITE (non-blocking) Writes out as much data as possible up to a maximum specified amount.

SEARCHTO$ (blocking) Reads in data (does not keep) until a search parameter is found. Non-matching data can be redirected to another port.

INPUT

If the variable is numeric and the value entered cannot be converted to a number, it writes as 0. This operation scans the data from left to right, shifting any number into the variable. It ignores any non-numeric character except the return character, which terminates the input, or Ctrl-C (**^C**) which terminates the program. The variable can be in string or numeric form.

Format

INPUT [<CHANNEL>:] <A$> [,<B$>]*

INPUT [<CHANNEL>:] <A>[,<B>]*

If the **[<channel>:]** is omitted, the default port, 0, will be used.

Parameters

<CHANNEL> = read data from this port. Default = 0.

<A,B,...,N> = variables to write.

When using multiple variables as targets, a corresponding number of lines are read. String and numeric variables can be intermixed.

Example This is an example of how to use the `INPUT` command:

```
10 OPEN #1: NAME "ZPL"
20 PRINT #1: "~HS"
30 FOR I = 1 TO 3
40 INPUT #1: A$
50 PRINT A$
60 NEXT I
```

In this example, a host status prints to the console after submitting the host status request **~HS** to the ZPL port. The Input/Output command of the ZBI interpreter is limited to the communications ports. File I/O is not supported.

INPUT ends processing a line with a **CR** or **LF**. This leads to a tricky situation. There are many ways different systems end a line: **CR**, **CRLF**, **LF**. If the ZBI program only uses **INPUT**, the next execution of the **INPUT** command will remove the extra **LF** or **CR**, in case of **LFCR**. However, if the program instead uses **INBYTE**, **DATAREADY** or the other commands, the extra **LF** will show up on the port. Here's a simple workaround to explicitly look for the **CRLF** that is in use:

SEARCHTO(<PORT>,CHR$(13)&CHR$(10),<INSTRING$>)

NOTE: The **INPUT** command does not accept control characters or the delete character. If these characters need to be processed, use the READ command.

Comments This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

If an invalid port is specified, **Error: Invalid port** is returned.

Example This shows the input command reading in multiple lines.

```
10 INPUT A$,B,C,D$,E$
```

Five lines would be read in: 3 strings and 2 numbers.

PRINT

This command sends data to the printer to be printed.

Format `PRINT [CHANNEL:] <expression> [,or; <expression>]* [;]`

Parameters

<CHANNEL> = write data to this port

<expression> = the value to write

The expression can be either a string or a numeric expression.

Using a , to separate expressions adds a space between them.

Using a ; to separate expressions does not put a space between them.

Using a ; at the end of a line ends the print statement without adding a new line (CR/LF).

Example This is an example of how to use the `PRINT` command:

```
10 LET A$ = "This is an example"
20 LET B$ = "of the PRINT Command."
30 PRINT A$, B$ ! adds a space between expressions
40 PRINT A$; B$ ! no space added
RUN
```

The result is:

This is an example of the PRINT Command.

This is an exampleof the PRINT Command.

Comments This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

OUTBYTE

This command outputs a byte to a port.

Format

OUTBYTE [<CHANNEL>:] <A>

OUTBYTE [<CHANNEL>:] <A$>

Parameters

<CHANNEL> = sends the byte to this port. Default = 0.

<A> = This is a numeric expression.

Values 0 through 255. If it is not within that range, it is truncated.

<A$> = This is the string expression. The first character is used. In the case of a NULL string, 0 is sent.

Example This is an example of how to use the `OUTBYTE` command:

```
LET A$="Hello"
OUTBYTE A$
```

This would only print the H character to the console.

```
OUTBYTE 4
```

This would print the control character **EOT** to the console. See an ASCII table for a list of the control characters.

Comments This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

INBYTE

This command forces the interpreter to pause until data is available. Use the **DATAREADY** function to determine if there is data on the port.

Format

INBYTE [<CHANNEL>:] <A>

INBYTE [<CHANNEL>:] <A$>

Parameters

<CHANNEL> = reads from this port. Default = 0.

<A> = integer value is set to the byte received.

<A$> = A single byte string is created with the byte received.The first character is used. In the case of a NULL string, 0 is sent.

Example This is an example of how to use the `INBYTE` to create an echo program:

```
10 INBYTE A$ !Takes one byte (char) from port #0
20 PRINT A$ !Prints the character to the console
30 GOTO 10
```

In this example, the interpreter pauses until the data is entered, then continues processing. This command enters all bytes in a string or integer, including control codes.

Comments **INBYTE** will block until a byte is received on the specified port. This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

READ

Description This is a non-blocking input function. It will read in all of the bytes available on the specified port.

Format `READ (<CHANNEL>, <A>, <MAXBYTES>)`

Parameters

<CHANNEL> = reads from this port. Default = 0.

<A$> = the string where the data will be placed

<MAXBYTES> = the maximum number of bytes to read

Returns The number of bytes read.

Example This is an example of the **READ** command:

```
1 CLOSE ALL
2 LET INPORT = CLIENTSOCKET("TCP","192.168.0.1",9100)
3 ON ERROR GOTO RECOVERY
4 LET WATERMARK = 5000
5 DO WHILE 1
6 IF LEN(DATA$) < WATERMARK THEN
7 LET BYTESREAD = READ(INPORT,DATA$,500)
8 ON ERROR GOTO RECOVERY
9 END IF
10 IF (LEN(DATA$) > 0) THEN
11 LET BYTES_WRITTEN = WRITE(INPORT,DATA$,LEN(DATA$))
12 ON ERROR GOTO RECOVERY
13 LET DATA$(1,BYTES_WRITTEN) = ""
14 END IF
15 IF BYTESREAD = 0 AND BYTESWRITTEN = 0 THEN
16 SLEEP 1 ! DON'T BOMBARD IF IDLE
17 END IF
18 LOOP
19 SUB RECOVERY
20 CLOSE #INPORT
```

WRITE

Description This is a non-blocking output function. It will write as many bytes as the output buffer can hold.

Format `WRITE (<CHANNEL>, <A>, <BYTES>)`

Parameters

<CHANNEL> = reads from this port. Default = 0.

<A$> = the string to write out.

<MAXBYTES> = The number of bytes to write

Returns The number of bytes written.

Example This is an example of **WRITE** command:

```
1 CLOSE ALL
2 LET INPORT = CLIENTSOCKET("TCP","192.168.0.1",9100)
3 ON ERROR GOTO RECOVERY
4 LET WATERMARK = 5000
5 DO WHILE 1
6 IF LEN(DATA$) < WATERMARK THEN
7 LET BYTESREAD = READ(INPORT,DATA$,500)
8 ON ERROR GOTO RECOVERY
9 END IF
10 IF (LEN(DATA$) > 0) THEN
11 LET BYTES_WRITTEN = WRITE(INPORT,DATA$,LEN(DATA$))
12 ON ERROR GOTO RECOVERY
13 LET DATA$(1,BYTES_WRITTEN) = ""
14 END IF
15 IF BYTESREAD = 0 AND BYTESWRITTEN = 0 THEN
16 SLEEP 1 ! DON'T BOMBARD IF IDLE
17 END IF
18 LOOP
19 SUB RECOVERY
20 CLOSE #INPORT
```

SEARCHTO$

This function performs a search until a specified string is found. The string the search yields is displayed.

Format

SEARCHTO$(A,B$)

SEARCHTO$(A,B$,C)

SEARCHTO$(A$,B$)

SEARCHTO$(A$,B$,C$)

Parameters

A = port number (0 to 9) to which requested data is sent

**A$** = string to search for B$

**B$** = string variable or string array. If B$ is an array, this command searches for all non-null strings in the B$ array.

C = a port in which the input is directed until B$ is found

**C$** = a string in which the characters in A$ are directed until B$ is found

Returns The string found.

Example This example shows how to use **SEARCHTO** to find a string on a port:

```
10 OPEN #1: NAME "SER"
20 LET A$ = SEARCHTO$(1,"^XA")
30 PRINT "FOUND:", A$
```

Example This example shows how to search for an array of strings:

```
10 OPEN #1: NAME "SER"
20 DECLARE STRING FIND$(3)
30 LET FIND$(1) = "ONE"
40 LET FIND$(2) = "TWO"
50 LET FIND$(3) = "THREE"
60 LET A$ = SEARCHTO$(1,FIND$)
70 PRINT "FOUND:", A$
```

Example This example shows unused data routed to a port.

```
10 OPEN #1: NAME "PAR"
20 OPEN #2: NAME "SER"
30 DECLARE STRING FIND$(3)
40 LET FIND$(1) = "ONE"
50 LET FIND$(2) = "TWO"
60 LET FIND$(3) = "THREE"
70 LET A$ = SEARCHTO$(1,FIND$,2)
80 PRINT "FOUND:", A$
```

Example This example shows how to use **SEARCHTO** to find a string within a string and direct the unused part of the string to another string:

```
10 LET A$ = "The faster you go, the shorter you are - Einstein"
20 LET B$ = SEARCHTO$(A$,"you", C$)
30 PRINT "FOUND:", B$
40 PRINT "DISCARDED:", C$
```

Comments **SEARCHTO** will block (wait) until the search string is found. If you want to be able to run other code while doing something similar, consider using **READ** with **POS**.

When using **SEARCHTO** with ports, it will block (wait) until the search string is found. If you want to be able to run other code while doing something similar, consider using **READ** to place data into a string. That string can be passed to **SEARCHTO** for processing.

Port Usage Examples

Before diving into the syntax of all the commands, let's look at some simple applications using the different features of the communications systems in ZBI.

Physical Ports (Serial, Parallel, USB, Bluetooth®)

Though the types of devices interacting with the printer's ports may vary greatly, internal to the printer, the ports are all handled in the same way. These ports are opened with the ZBI **OPEN** command and closed with the ZBI **CLOSE** command. When one of these ports is opened, it is disconnected from the ZPL parser and any data in the buffer will be redirected to the ZBI environment.

Example In the following example, "**SER**" could be replaced by "**PAR**", "**USB**", or "**BLU**" depending on the application.

```
10 CLOSE ALL
20 LET INPORT = 1
25 SLEEP 1
30 OPEN #INPORT: NAME "SER"
35 ON ERROR GOTO 25
40 PRINT #INPORT: "Enter your name:";
50 INPUT #INPORT: YOURNAME$
55 ON ERROR GOTO 70
60 PRINT #INPORT: "You entered: "; YOURNAME$
70 CLOSE #INPORT
```

ZPL Parser

To make a ZBI program print, it is necessary to create a connection from the program to the ZPL parser on the printer. The connection will function in the same way as a connection to a physical port, except that the connection will not automatically terminate. The ZPL parser in the printer can handle many incoming connections simultaneously. For example, a ZBI program could take control of the serial port and send label formats to the ZPL parser, while the parallel port (unopened by ZBI) could also be used to send label formats directly into the parser.

NOTE: The ZPL parser will lock onto one port once a format is started (via the **^XA** command). So, in some cases, is it desirable to start and stop your communications to ZPL in one continuous sequence.

Another use of ZBI is to check printer status, while another application prints to another port.

Example Here is how that can be done:

```
10 OPEN #1: NAME "ZPL"
20 PRINT #1: "~HS"
30 FOR I = 1 TO 3
40 INPUT #1: A$
50 PRINT A$
60 NEXT I
```

TCP Client

There are two methods for making a TCP connection to another server. The first method uses the **OPEN** command while the second method uses the **CLIENTSOCKET** method.

CLIENTSOCKET is the preferred method.

Example The following example demonstrates this method:

```
10 CLOSE ALL
20 LET INPORT = CLIENTSOCKET("TCP","192.168.0.1",9100)
40 LET OUTSTR$ = "REQUESTING SERVER NAME";
50 DO WHILE (LEN(OUTSTR$) > 0)
60 LET BYTES_WRITTEN = WRITE(INPORT,OUTSTR$,LEN(OUTSTR$))
70 ON ERROR GOTO RECOVERY
80 LET OUTSTR$ = OUTSTR$(1+BYTES_WRITTEN:LEN(OUTSTR$))
90 LOOP
100 INPUT #INPORT: YOURNAME$
110 PRINT #INPORT: "Server returned: "; YOURNAME$
120 CLOSE #INPORT
130 SUB RECOVERY
140 END
```

TCP Server

Setting up a listening server in the printer can be accomplished with the **SERVERSOCKET** function. To connect to incoming TCP sessions, use the **ACCEPT** function.

When starting the application, call **SERVERSOCKET**. This function will create a handle for this listening server. Check for incoming connections at regular intervals with the **ACCEPT** function. If there are no pending sessions, the **ACCEPT** function will return with an error. Handle the error using the **ON ERROR** command and continue looking for other sessions later.

Depending on how the program is set up, it is possible to handle one or more sessions at a time. If the program is configured to allow only one session, the other connections will remain pending until they are shut down by the requesting client or the ZBI program connects them.

Example Here is an example of the **SERVERSOCKET and ACCEPT** commands:

```
10 CLOSE ALL
20 LET SERVER_HANDLE = SERVERSOCKET("TCPX",19100)
30 REM There are no connections yet we are just listening for them
40 REM Lets loop until we get a connection
50 SLEEP 1
60 LET INPORT = ACCEPT(SERVER_HANDLE,CLIENT_INFO$)
70 ON ERROR GOTO 50
80 PRINT #INPORT: "You have successfully connected!"
90 PRINT #INPORT: "Login:";
100 INPUT #INPORT: LOGIN$
110 PRINT #INPORT: "Password:";
120 INPUT #INPORT: PASSWORD$
130 REM We will not be nice and reject the connection
130 PRINT #INPORT: "Login failed"
140 CLOSE #INPORT
150 GOTO 60 ! Go look for the next connection
160 END
```

UDP Client

There are also two methods for making a UDP connection to another server. The first method uses the **OPEN** command, while the second method uses the **CLIENTSOCKET** method. UDP is a one way communication medium, thus, you can only use output commands. Because UDP is connectionless, the output will be queued up until an **EOT** character is written or the maximum packet size is exceeded. Once the **EOT** character is written, the packet is formatted and sent.

With UDP, it is important to be careful about understanding what the network being used will support.

In many cases, there will be a limit to the size of the packet that can be used, typically between 1000 and 1500 bytes, but some networks cut this down into the 500 to 600 byte range. To be safe, keep your packets less than 500 bytes.

UDP does not guarantee transmission. See UDP specifications for more details.

Example Since **CLIENTSOCKET** is the preferred method, an example is shown below.

```
10 CLOSE ALL
20 LET INPORT = CLIENTSOCKET("UDP","192.168.0.1",22222)
30 LET EOT$ = CHR$(4)
40 PRINT #INPORT: "Packet #"; I; EOT$;
50 LET I = I + 1
60 SLEEP 1
70 GOTO 40
```

UDP Server

Setting up a listening server in the printer can be accomplished with the **SERVERSOCKET** function. Then, to connect to incoming UDP packets, use the function **ACCEPT**. When starting your application, call **SERVERSOCKET**. This function will create a handle for this listening server. Check for incoming packets at a regular interval with the **ACCEPT** function. If there are no pending sessions, the **ACCEPT** function will return with an error. Just handle the error using the **ON ERROR** command and continue looking for other sessions later. You will need to call **ACCEPT** for each incoming packet. When the accept is successful, all of the data will be available. Call **READ** with a **MAX** string size of 2000 and you will have the whole packet in your string. Close the port and wait for the next packet. You can only read in data using a UDP server.

Example Here is an example of how to set up to receive UDP messages:

```
10 CLOSE ALL
20 LET ZPLPORT = 1
35 OPEN #ZPLPORT: NAME "ZPL"
40 LET SERVER_HANDLE = SERVERSOCKET("UDP",33333)
50 REM There are no connections yet: listening
60 REM Let’s loop until we get a connection
70 SLEEP 1
80 LET INPORT = ACCEPT(SERVER_HANDLE,CLIENT_INFO$)
90 IF INPORT = -1 THEN
92 GOTO 70
94 END IF
100 LET PACKET_SIZE = READ(INPORT,PACKET$,2000)
110 PRINT #ZPLPORT: "^XA^FO100,100^A0N,40,40^FDPACKET FROM:";
115 PRINT #ZPLPORT: CLIENT_INFO$; "^FS"
120 PRINT #ZPLPORT: "^FO100,150^A0N,40,40^FDPACKET SIZE:";
125 PRINT #ZPLPORT: PACKET_SIZE; "^FS"
130 PRINT #ZPLPORT: "^FO100,200^A0N,40,40^FDPACKET DATA:";
135 PRINT #ZPLPORT: PACKET$; "^FS^XZ"
140 CLOSE #INPORT
150 GOTO 60 ! go look for the next connection
160 END
```

E-mail

ZBI can be used to enhance the printer’s ability to send status via e-mail messages. The process is simple: open the email port "EML", send the recipient list, send the header, and send the body of the message.

The printer can only process a limited number of outgoing email messages at one time. For this reason, error handling should be used when opening the connection to wait for the printer to be ready to send the message. The EOT character is important for delimiting sections of the email message. If it is left out, the message will not be sent properly.

Before the following code will work, the email settings for the print server must be set up. Consult the print server manual to learn how to configure the unit.

Example Here is an example of how to send e-mails:

```
1 REM EOT$ this is used to denote end of transmission
5 LET EOT$ = CHR$(4)
```

```
1 REM Open a connection to the e-mail port and if it errors
1 REM try again until complete
10 OPEN #1: NAME "EML"
15 ON ERROR GOTO 10
1 REM Specify address to send message to then end signal end
1 REM of recipients with EOT$
1 REM To send to multiple addressees separate addressees by
1 REM space
20 PRINT #1: "youraddress@yourdomain.com";EOT$;
1 REM Fill in the message information
30 PRINT #1: "From: HAL"
40 PRINT #1: "To: Dave"
50 PRINT #1: "Subject: A message from HAL"
60 PRINT #1: ""
70 PRINT #1: "Dave, I am sorry I can not let you do that."
80 PRINT #1: i
1 REM Terminate message
90 PRINT #1: "";EOT$
1 REM You must close the port, each open port is only good
1 REM for sending one message
100 CLOSE #1
```

File System

This section shows how programs and formats can be saved and recalled. Here's a quick list of these commands:

STORE Saves the program currently in memory as the specified file name.

LOAD Transfers a program file previously stored in the printer's memory and opens it in the ZBI Program Memory.

DIR With no filter included, prompts the printer to list all of the ZBI programs residing in all printer memory locations.

DELETE Removes a specified file from the printer's memory.

Runtime Access

The following example is a method to store runtime data in the printer memory. The file system in the printer is limited to writing one file at a time. Since only one component of the printer can have write access to the file system, the ZPL parser is the component with this access. For ZBI to use the ZPL parser as a gateway into printer memory, the ZPL comment command (**^FX**) is used.

Example

```
AUTONUM 1,1
REM ******* TEST FOR SUBROUTINES **********************
LET ZPLPORT = 1 OPEN #ZPLPORT: NAME "ZPL"
LET SIZE = 5
LET FILENAME$ = "R:TESTSYS.ZPL"
DECLARE STRING DATAIN$(SIZE)
LET DATAIN$(1) = "ONE"
LET DATAIN$(2) = "TWO"
LET DATAIN$(3) = "THREE"
LET DATAIN$(4) = "FOUR"
LET DATAIN$(5) = "FIVE"
GOSUB STOREDATA
GOSUB GETDATA
FOR I = 1 TO SIZE
IF DATAIN$(I) <> DATAOUT$(I) THEN
PRINT #ZPLPORT: "^XA^FO100,100^A0N,50,50^FDERROR:";
PRINT #ZPLPORT: DATAOUT$(I);"^XZ"
END IF
NEXT I
END
REM **** SUBROUTINE STOREDATA *************************
REM INPUT: ZPLPORT, DATAIN$, SIZE, FILENAME$ ***********
SUB STOREDATA
PRINT #ZPLPORT: "^XA^DF" & FILENAME$ & "^FS"
PRINT #ZPLPORT: "^FX"; SIZE; "^FS"
FOR I = 1 TO SIZE
PRINT #ZPLPORT: "^FX" & DATAIN$(I) & "^FS"
NEXT I
PRINT #ZPLPORT: "^XZ"
RETURN
REM **** SUBROUTINE GETDATA - *************************
REM INPUT: ZPLPORT, FILENAME$ *************************
REM ** OUTPUT: DECLARES AND FILLS DATAOUT$ AND FILLS SIZE
SUB GETDATA
PRINT #ZPLPORT: "^XA^HF" & FILENAME$ & "^XZ"
SLEEP 1
LET RESULT$ = ""
```

```
FOR J = 1 TO 25
LET A = READ(ZPLPORT,TEMP$,5000)
LET RESULT$ = RESULT$ & TEMP$
IF POS(RESULT$,"^XZ") <> 0 THEN
EXIT FOR
END IF
SLEEP 1
NEXT J
LET RESULT$(1:POS(RESULT$,"^FX")+2) = ""
LET SIZE = VAL(EXTRACT$(RESULT$,"","^"))
DECLARE STRING DATAOUT$(SIZE)
FOR I = 1 TO SIZE
LET RESULT$(1:POS(RESULT$,"^FX")+2) = ""
LET DATAOUT$(I) = EXTRACT$(RESULT$,"","^")
NEXT I
LET RESULT$ = ""
LET TEMP$ = ""
RETURN
```

STORE

This command saves the program currently in memory as the specified file name. The format listed below is used.

Format **STORE <filename$>**

Parameters **<filename$>** = the name of the file to be stored. Drive location and file name must be in quotation marks.

Example This is an example of how to use the **STORE** command:

```
STORE "E:PROGRAM1.BAS"
```

Comments For a file name to be valid, it must conform to the 8.3 Rule: each file must have no more than eight characters in the file name and have a three-character extension. Here the extension is always **.BAS** (for example, **MAXIMUM8.BAS**).

This is an interactive command that takes effect as soon as it is received by the printer.

The ZBI-Developer IDE will take care of this for you with the **SEND TO** option on your program.

LOAD

This command transfers a program file previously stored in the printer's memory and opens it in the ZBI Program Memory.

If the program file does not exist, the ZBI Program Memory is cleared and no program is opened.

Format **LOAD <filename$>**

Parameters **<filename$>** = the file name to be loaded into memory. Drive location and file name must be in quotation marks. If the drive location is not specified, all drives will be searched.

Example Here are examples of how to use the LOAD command:

```
LOAD "PROGRAM1.BAS"
LOAD "E:PROGRAM1.BAS"
```

Comments This is an interactive command that takes effect as soon as it is received by the printer.

DIR

This command, with no filter included, prompts the printer to list all of the ZBI programs residing in all printer memory locations.

Including a filter signals the printer to limit the search; including a drive location signals the printer to search in only one location.

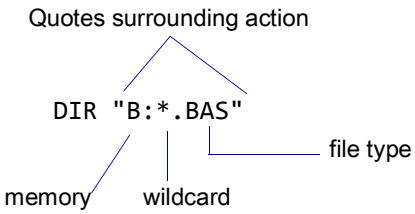
Asterisks (*) are used as wild cards. A wild card (*) finds every incidence of a particular request. The example here, **`DIR "B:*.BAS"`**, signals the printer to search for every file with a **`.BAS`** extension in B: memory.

Format **`DIR [<filter$>]`**

Parameters **`[<filter$>]`** = the name of the file to be accessed (optional). Drive location and file name must be in quotation marks.

Default = **`"*:*.bas"`**

IMPORTANT: Quotes must be around what you are doing. This shows you how to use the wildcard (**`*`**) to search for all **`.BAS`** files in **`B:`** memory:



Example N/A

Comments This is an interactive command that takes effect as soon as it is received by the printer.

DELETE

This command removes a specified file from the printer's memory.

Format **`DELETE <filename$>`**

Parameters **`<filename$>`** = the name of the file to be deleted. Drive location and filename must be in quotation marks.

Example This is an example of deleting a specified file from printer memory:

```
DELETE "E:PROGRAM1.BAS"
```

Comments This is an interactive command that takes effect as soon as it is received by the printer.

Comma Separated Values (CSV)

Accessing Comma Separated Value (CSV) and Text File Functions

This section describes the functions to access CSV files and ASCII plain-text files. Here is a quick list of these commands:

CSVLOAD Loads the contents of a CSV file in a two dimensional string array.

CSVSTORE Stores the contents of a two dimensional string array in a CSV file.

TXTLOAD Loads the contents of an ASCII plain-text file into a string variable.

TXTSTORE Stores the contents of a string variable in an ASCII plain text file.

CSVLOAD

This function will load the delimited values from a CSV file, defined by **FILENAME$**, and store them in the two-dimensional array, **DEST$**.

Format

CSVLOAD(DEST$, FILENAME$)

CSVLOAD(DEST$, FILENAME$, DELIM$)

Parameters

DEST$** = two dimensional array that will hold the rows and columns from the CSV file specified by the **FILENAME$ variable. If there is not enough room in **DEST$**, or if it has the wrong size, it will be changed to fit the data from the file. The data originally in **DEST$** will be overwritten.

FILENAME$ = name of the file to load. Drive location and file name must be in quotation marks. The file extension must be either "**.CSV**" or "**.TXT**".

DELIM$** = optional delimiter that is used in the CSV file instead of a comma. If **DELIM$ is not provided a comma will be used by default. The delimiter must be a single character that is not a quote, carriage return, or newline.

Returns The number of elements in each row of the CSV file. The function will return 0 if errors were detected in the CSV file, or if the file could not be read.

Example This example shows how to print the values in a CSV file with a comma delimiter.

```
10  DECLARE STRING CSVDB$(1,2)
20  LET FILENAME$ = "E:RECORDS.CSV"
30  LET NUMOFCOLS = CSVLOAD(CSVDB$, FILENAME$)
40  LET NUMOFROWS = ROWSIZE(CSVDB$)
100 FOR I = 1 TO NUMOFROWS STEP 1
110     FOR J = 1 TO NUMOFCOLS STEP 1
120         PRINT CSVDB$(I, J), " ";
200     NEXT J
210     PRINT ""
300 NEXT I
```

Example This example shows how to print the values in a CSV file that uses a '|' as a delimiter.

```
10  DECLARE STRING CSVDB$(1,2)
20  LET FILENAME$ = "E:EMPLOYEE.CSV"
30  LET NUMOFCOLS = CSVLOAD(CSVDB$, FILENAME$, "|")
40  LET NUMOFROWS = ROWSIZE(CSVDB$)
100 FOR I = 1 TO NUMOFROWS STEP 1
110     FOR J = 1 TO NUMOFCOLS STEP 1
120         PRINT CSVDB$(I, J), " ";
200     NEXT J
210     PRINT ""
300 NEXT I
```

Comments The maximum CSV file size supported will vary based upon available RAM within the printer.

CSV File Information

The file format should follow the rules in IETF RFC 4180: http://tools.ietf.org/html/rfc4180

The maximum number of columns per row in a CSV file is 256.

Each row must be 2048 characters or less including the delimiter. The carriage return/line feed (CRLF) does not count toward the limit.

Each row in the CSV file must have the same number of elements. If there are any missing elements in the CSV file (indicated by two adjacent commas or a comma at the end of a row), they will be represented as empty strings.

If an element in the CSV file contains a quote, it should be represented as two quotes. Additionally, if an element contains a quote, a new line, a carriage return, or the delimiter character, the element must be within quotes. For example, a value that is used to store a measurement in feet and inches (4' 5") must be formatted as "4' 5""" within the CSV file.

CSVSTORE

This function will store the values of a two dimensional array into a CSV file on the file system. Each element within the array is treated as a single value within the CSV file.

Format

```
CSVSTORE(SRC$, FILENAME$)
CSVSTORE(SRC$, FILENAME$, DELIM$)
```

Parameters

SRC$ = two dimensional array of strings to be written to a CSV file.

FILENAME$ = name of the file to store the array contents. Drive location and file name must be in quotation marks. The file extension must be either ".**CSV**" or ".**TXT**".

**DELIM$** = optional delimiter that is used in the CSV file instead of a comma. If DELIM$ is not provided a comma will be used by default. The delimiter must be a single character that is not a quote, carriage return, or newline.

Returns A 0 if there were no errors. A 1 is returned if **SRC$** is not a string array, if the file could not be written, or if **SRC$** contains errors that prevent the file from being stored.

Example This example shows how to convert a comma delimited CSV file into a "^" delimited TXT file and print the contents.

```
10  DECLARE STRING CSVDB$(1,2)
20  LET NUMOFCOLS = CSVLOAD(CSVDB$, "E:RECORDS.CSV")
30  LET CSVERROR = CSVSTORE(CSVDB$, "E:NEWREC.TXT", "^")
40  LET NUMOFCOLS = CSVLOAD(CSVDB$, "E:NEWREC.TXT", "^")
50  LET NUMOFROWS = ROWSIZE(CSVDB$)
100 FOR I = 1 TO NUMOFROWS STEP 1
110     FOR J = 1 TO NUMOFCOLS STEP 1
120         PRINT CSVDB$(I, J), " ";
200     NEXT J
210     PRINT ""
300 NEXT I
```

Comments The elements of the array should follow the rules in IETF RFC 4180: http://tools.ietf.org/html/rfc4180

There is no limit on the number of columns per row when storing to a CSV file. However, a file stored with rows that exceed the column limit imposed by **CSVLOAD** will not be loaded by the **CSVLOAD** function.

There is no limit on the size of a row when stored to a CSV file. However, a file stored with rows that exceed the size limit imposed by **CSVLOAD** will not be loaded by the **CSVLOAD** function.

TXTLOAD

This function will read the contents of an ASCII text file into a ZBI string variable.

Format `TXTLOAD(DEST$, FILENAME$)`

Parameters

DEST$** = string to store the contents of **FILENAME$.

FILENAME$ = name of the file to read. Drive location and file name must be in quotation marks. The file extension must be either ".**CSV**" or ".**TXT**".

Returns The number of bytes read from the file. The function will return 0 if the file could not be read.

Example This example shows how to print out the contents of a file.

```
10 LET TXTSIZE = TXTLOAD(TXTDATA$, "E:MYDATA.TXT")
20 PRINT STR$(TXTSIZE), "bytes:", TXTDATA$
```

Comments The data originally in **DEST$** will be overwritten upon completion of this function.

TXTSTORE

This function will store the contents of a ZBI string in an ASCII text file.

Format `TXTSTORE(SRC$, FILENAME$)`

Parameters

SRC$** = string to store to **FILENAME$.

FILENAME$ = name of the file to store. Drive location and file name must be in quotation marks. The file extension must be either ".**CSV**" or ".**TXT**".

Returns Returns a 0 if there were no errors, otherwise a 1 is returned.

Example This example shows how to append a text file.

```
10 LET TXTSIZE = TXTLOAD(TXTDATA$, "E:MYDATA.TXT")
11 REM Append a date/time stamp to the file
20 LET TXTDATA$ = TXTDATA$ & " " & DATE$ & " " & TIME$
30 LET TXTSIZE = TXTSTORE(TXTDATA$, "E:MYDATA.TXT")
40 PRINT TXTDATA$
```

Events

This section explains how to capture and trigger internal events in the printer. Here's a quick list of these commands:

Available Events A table that correlates a ZBI event with an identification number.

ZBI Key Names Details the names of each printer's front panel buttons, ZBI names, and ZBI event ID.

REGISTEREVENT Sets up the **HANDLEEVENT** function to receive notification when the specified event has occurred.

UNREGISTEREVENT Allows events that are currently set to be captured by the program to no longer be captured.

HANDLEEVENT Once events have been registered, this function is used to see what events have occurred.

TRIGGEREVENT Allows for front panel buttons to be triggered programatically.

There are certain events in the printer that a ZBI 2.0 program can receive. To do this, the program first registers for the event. On a regular basis, call a function to handle events. When an event occurs that the program is registered for, the function will return the event's identification number.

Available Events

ZBI Event ID	ZBI Event
3	menu key
4	pause key
5	feed key
6	cancel key
7	up arrow key
8	plus key
9	minus key
10	enter key
11	setup exit key
12	select key
13	cancel all event
14	config label
15	timer1
16	timer2
17	timer3
18	timer4
19	timer5
20	spare unused
21	previous key
22	next save key

ZBI Event ID	ZBI Event
23	calibrate key
24	paper out set
25	paper out clear
26	ribbon out set
27	ribbon out clear
28	head too hot set
29	head too hot clear
30	head cold set
31	head cold clear
32	head open set
33	head open clear
34	supply too hot set
35	supply too hot clear
36	ribbon in set
37	ribbon in clear
38	rewind full set
39	rewind full clear
40	cutter jammed set
41	cutter jammed clear
42	paused set
43	paused clear
44	pq completed set
45	pq completed clear
46	label ready set
47	label ready clear
48	head element bad set
49	head element bad clear
50	basic runtime set
51	basic runtime clear
52	basic forced set
53	basic forced clear
54	power on set
55	power on clear
56	clean printhead set
57	clean printhead clear
58	media low set
59	media low clear
60	ribbon low set
61	ribbon low clear

ZBI Event ID	ZBI Event
62	**replace head set**
63	**replace head clear**
64	**battery low set**
65	**battery low clear**
66	**rfid error set**
67	**rfid error clear**
68	**any messages set**
69	**any messages clear**
70	auto baud
71	factory default
72	networking default
73	networking factory
74	print width
75	darkness adjust
76	calibrate
77	scroll key
78	soft key 1
79	soft key 2

ZBI Key Names

This section details the names to use for each printer's front panel buttons when creating ZBI 2.0 programs to capture the buttons.

ZT200/ZT400/ZT500/ZT600/ZD500/QIn

ZT2X0	ZT400/ ZT500/ ZT600	ZD500	QIn	ZBI Event ID	ZBI Name
Left Soft button				76	`soft key 1`
Right Soft Button				77	`soft key 2`
Plus	Up Arrow			6	`plus key`
Minus	Down Arrow			7	`minus key`
Left Arrow				19	`previous key`
Right Arrow				20	`next save key`
Setup	OK	Check	OK	10	`select key`
Pause			no key	2	`pause key`
Feed				3	`feed key`
Cancel			no key	4	cancel key

Xi4/RXi4/XiIIIPlus/PAX4/105SL/ZE500

XiIIIPlus/PAX4/Xi4/RXi4/ ZE500/105SL Plus Front Panel Key	105SL Front Panel Key	ZBI Event ID	ZBI Name
Right Oval	Plus (+)	6	`plus key`
Left Oval	Minus (-)	7	`minus key`
Previous		19	`previous key`
Next/Save		20	`next save key`
Setup/Exit		9	`setup exit key`
Pause		2	`pause key`
Feed		3	`feed key`
Cancel		4	`cancel key`
Calibrate		21	`calibrate key`

HC100

Front Panel Key	ZBI Event ID	ZBI Name
Pause	2	`pause key`
Feed	3	`feed key`
Eject		`eject key`

ZM400/ZM600/RZ400/RZ600/Z4Mplus/Z6Mplus

Front Panel Key	ZBI Event ID	ZBI Name
Feed	3	`feed key`
Pause	2	`pause key`
Cancel	4	`cancel key`
Setup/Exit	9	`setup exit key`
Select	10	`select key`
Plus (+)	6	`plus key`
Minus (-)	7	`minus key`

S4M

Front Panel Key	ZBI Event ID	ZBI Name
Menu	1	`menu key`
Enter	8	`enter key`
Cancel	4	`cancel key`
Feed	3	`feed key`
Pause	2	`pause key`
Left Arrow	4	`cancel key`
Right Arrow	3	`feed key`
Up Arrow	5	`up arrow key`
Down Arrow	2	`pause key`

G-Series

Front Panel Key	ZBI Event ID	ZBI Name
Feed key	3	`Feed key`
Select key	10	`Select key`
Scroll key	75	`Scroll key`

KR403 / 2824 Plus Series

Front Panel Key	ZBI Event ID	ZBI Name
Feed key	3	`Feed key`

REGISTEREVENT

Description This function will set up the **HANDLEEVENT** function to receive notification when the specified event has occurred. Events can be registered for one time or until the program is exited.

IMPORTANT: If an event occurs twice or more before the **HANDLEEVENT** function is called, only one event will be received.

Format

```
REGISTEREVENT(X)
REGISTEREVENT(X,Y)
REGISTEREVENT(X,Y,Z)
```

Parameters

(X) = This is the ID of the event being registered for.

(Y) = If Y=1: the event happens once; If Y=0: the event stays registered for the duration of the program, or until it is unregistered.

(Z) = For System Events: if Z=0, the event will still be handled by the printer. If Z=1, then only ZBI will receive the event.

For Timer Events: this is the timer interval in mSec. If the interval is less than 0 or greater than 1,000,000,000, it is set to 1000.

Returns The ID of the successfully registered event. If an event was not successfully registered, a -1 is returned.

Example Here is an example of how to use the REGISTEREVENT command:

```
1 REM This example shows how to override the functionality of the feed
1 REM key
1 REM using the event system. After all why waste a label when you
1 REM could put
1 REM valuable information there
AUTONUM 1,1
CLOSE ALL
LET ZPLPORT = 1
OPEN #ZPLPORT: NAME "ZPL"
LET FEEDKEY = 3
LET TMP = REGISTEREVENT(FEEDKEY, 0, 1)
DO WHILE 1 = 1
LET EVT = HANDLEEVENT()
IF EVT = FEEDKEY THEN
GOSUB PRINTINFO
END IF
SLEEP 1
LOOP
REM **** SUBROUTINE PRINTINFO *** expects ZPLPORT *******
SUB PRINTINFO
PRINT #ZPLPORT: "^XA"
PRINT #ZPLPORT: "^FO30,30^A0N,50,50^FDZebra Technologies^FS"
PRINT #ZPLPORT: "^FO30,85^A0N,35,35^FDwww.zebra.com^FS"
PRINT #ZPLPORT: "^FO30,125^A0N,35,35^FDsupport.zebra.com^FS"
PRINT #ZPLPORT: "^FO30,165^A0N,35,35^FDFW Version: "
PRINT #ZPLPORT: GETVAR$("appl.name") & "^FS"
PRINT #ZPLPORT: "^FO30,205^A0N,35,35^FDPrinter Unique ID:"
PRINT #ZPLPORT: GETVAR$("device.unique_id") & "^FS"
PRINT #ZPLPORT: "^FO30,245^A0N,35,35^FDActive Network: "
PRINT #ZPLPORT: GETVAR$("ip.active_network") & "^FS"
PRINT #ZPLPORT: "^FO30,285^A0N,35,35^FDZBI Memory Usage: "
PRINT #ZPLPORT: GETVAR$("zbi.start_info.memory_alloc") & "^FS"
PRINT #ZPLPORT: "^FO30,325^A0N,35,35^FDOdometer: "
PRINT #ZPLPORT: GETVAR$("odometer.total_print_length") & "^FS"
PRINT #ZPLPORT: "^XZ"
```

Comments None

UNREGISTEREVENT

Description This function allows events that are currently set to be captured by the program to no longer be captured. Once called events will return to the normal method of processing if the REGISTEREVENT function Z parameter was set to 1.

Format `UNREGISTEREVENT(X)`

Parameters `(X)` = the ID of the event to stop

Returns 0 if the event is a valid event to unregister. A -1 if the event does not exist.

Example Here is an example of how to use the `UNREGISTEREVENT` command:

```
AUTONUM 1,1
LET OUTSTR$ = "Processing"
LET LOOPCTR = 200
LET TIMER5 = 17
LET TMP = REGISTEREVENT(TIMER5, 0, 1000)
DO WHILE LOOPCTR > 0
LET EVT = HANDLEEVENT()
IF EVT = TIMER5 THEN
LET A = SETVAR("device.frontpanel.line2",OUTSTR$)
LET OUTSTR$ = OUTSTR$ & "."
IF LEN(OUTSTR$) >16 THEN
LET OUTSTR$ = "Processing"
END IF
END IF
LET LOOPCTR = LOOPCTR - 1
SLEEP 1
LOOP
LET TMP = UNREGISTEREVENT(TIMER5)
LET A = SETVAR("device.frontpanel.line2","")
END
```

Comments None

HANDLEEVENT

Description Once events have been registered, this function is used to see what events have occurred.

Format `HANDLEEVENT()`

Parameters N/A

Returns The ID of the event that occurred. One event at a time will be returned through this function. The order of the events are based on priority. The priority is based on the ID number of the event, with the exception of the timer events, which have the highest priority.

Example Here are examples of how to use the `HANDLEEVENT` command:

```
1 REM This example shows how to override the feed key functionality
1 REM using the event system. Why waste a label when you could put
1 REM valuable information there
AUTONUM 1,1
CLOSE ALL
LET ZPLPORT = 1
OPEN #ZPLPORT: NAME "ZPL"
LET FEEDKEY = 3
LET TMP = REGISTEREVENT(FEEDKEY, 0, 1)
DO WHILE 1 = 1
LET EVT = HANDLEEVENT()
IF EVT = FEEDKEY THEN
GOSUB PRINTINFO
END IF
SLEEP 1
LOOP
REM ******** SUBROUTINE PRINTINFO ***
REM *** expects ZPLPORT *************
SUB PRINTINFO
PRINT #ZPLPORT: "^XA"
PRINT #ZPLPORT: "^FO30,30^A0N,50,50";
PRINT #ZPLPORT: "^FDZebra Technologies^FS"
PRINT #ZPLPORT: "^FO30,85^A0N,35,35";
PRINT #ZPLPORT: "^FDwww.zebra.com^FS"
PRINT #ZPLPORT: "^FO30,125^A0N,35,35";
PRINT #ZPLPORT: "^FDsupport.zebra.com^FS"
PRINT #ZPLPORT: "^FO30,165^A0N,35,35";
PRINT #ZPLPORT: "^FDFW Version: ";
PRINT #ZPLPORT: GETVAR$("appl.name") & "^FS"
PRINT #ZPLPORT: "^FO30,205^A0N,35,35";
PRINT #ZPLPORT: "^FDPrinter Unique ID:";
PRINT #ZPLPORT: GETVAR$("device.unique_id") & "^FS"
PRINT #ZPLPORT: "^FO30,245^A0N,35,35";
PRINT #ZPLPORT: "^FDActive Network: ";
PRINT #ZPLPORT: GETVAR$("ip.active_network") & "^FS"
PRINT #ZPLPORT: "^FO30,285^A0N,35,35";
PRINT #ZPLPORT: "^FDZBI Memory Usage: ";
PRINT #ZPLPORT: GETVAR$("zbi.start_info.memory_alloc") & "^FS"
PRINT #ZPLPORT: "^FO30,325^A0N,35,35";
PRINT #ZPLPORT: "^FDOdometer: ";
PRINT #ZPLPORT: GETVAR$("odometer.total_print_length") & "^FS"
PRINT #ZPLPORT: "^XZ"
```

Comments None

TRIGGEREVENT

This function allows for front panel buttons to be triggered programatically.

Format `TRIGGEREVENT(X)`

Parameters

X = the ID of the event from the possible event list to TRIGGER.

See the following printer tables for events that can be triggered by this command:

- Xi4/RXi4/XiIIIPlus/PAX4/105SL/ZE500 on page 506
- 105SL Front Panel Key on page 506
- ZM400/ZM600/RZ400/RZ600/Z4Mplus/Z6Mplus on page 507
- S4M on page 507

Returns Always returns 0.

Example Here are examples of how to use the **`TRIGGEREVENT`** command:

```
1 REM THIS IS AN EXAMPLE OF HOW TO TRIGGER AN EVENT
AUTONUM 1,1
LET PAUSEKEY = 2
DO WHILE 1 = 1
LET A = TRIGGEREVENT(PAUSEKEY)
LET A = SETVAR("device.frontpanel.line2",str$(A))
SLEEP 2
LOOP
```

Comments None

Systems

This section contain miscellaneous systems interface functions. Here's a quick list of these commands:

ISERROR Returns a non-zero value if there is an internal error set in the printer.

ISWARNING Returns a non-zero value if there is an internal warning set in the printer.

SLEEP Specifies the time that the interpreter pauses.

SETERR Sends a message to the printer to set the error flag.

CLRERR Sends a message to the printer to clear the error flag.

ON ERROR Prevents a program from halting in the event of an error.

ISERROR

This function returns a non-zero value if there is an internal error set in the printer. Otherwise, the numeral returned will 0.

Format `ISERROR`

Parameters N/A

Returns 0 for no errors; 1 if there is an error.

Example Here is an example of the **`ISERROR`** command.

```
10 PRINT ISERROR
RUN
0
```

Comments None

ISWARNING

This function returns a non-zero value if there is an internal warning set in the printer. Otherwise, the numeral returned will 0.

Format `ISWARNING`

Parameters N/A

Returns 0 for no errors; 1 if there is an error.

Example Here is an example of the `ISWARNING` command.

```
10 PRINT ISWARNING
RUN
0
```

Comments None

SLEEP

This command specifies the time that the interpreter pauses. This command could be sent to the printer after sending a label format to be printed. The interpreter pauses in its processing for the amount of time specified.

Format `SLEEP <A>`

Parameters `<A>` = the time in seconds (0 to 500) the interpreter pauses.

Example This is an example of how to use the **SLEEP** command:

```
10 SLEEP 450
```

Comments If a timer is needed, use the **Event** system. The timer will allow for processing other items, where **SLEEP** will stop execution of any ZBI commands for the specified **SLEEP** period.

This is a program command and must be preceded by a line number.

Calling **SLEEP** with **<A>** set to zero will force the ZBI task to yield to the rest of the system and allow any pending tasks to run (e.g., pending ZPL commands). If there are no pending tasks, ZBI will sleep for a minimum of 8 milliseconds.

SETERR

This command sends a message to the printer to set the error flag. A logical interpreter flag is triggered in the printer. This error is referenced as a BASIC Forced Error.

Format `SETERR`

Parameters N/A

Example An example of the **SETERR** and **CLRERR** commands.

```
AUTONUM 1,1
OPEN #1:NAME "ZPL"
PRINT #1: "^XA^SXO,A,Y,Y^XZ"
CLOSE #1
FOR I=1 TO 10
SLEEP 5
IF MOD(I,2)=1 THEN
SETERR
ELSE
CLRERR
ENDIF
NEXT I
```

Comments This is a program command and must be preceded by a line number.

CLRERR

This command sends a message to the printer to clear the error flag. A logical interpreter flag is cleared in the printer. This error is referenced as a BASIC Forced Error.

Format `10 CLRERR`

Parameters N/A

Example See SETERR on page 515.

Comments This is a program command that is preceded by a line number.

ON ERROR

The **ON ERROR** command can be used to prevent a program from halting in the event of an error. If an error occurs in a previous line during program execution, the **ON ERROR** statement calls the **GOTO** or **GOSUB** statement and allows the program to continue.

Format

```
ON ERROR GOTO <A>
ON ERROR GOSUB <A>
```

Parameters **<A>** = the destination location in the program should an error be triggered on the previous line.

Example This is an example of how to use the `ON ERROR` command:

```
30 LET A = B/C
40 ON ERROR GOTO 100
...
100 PRINT "DIVIDE BY ZERO OCCURRED"
110 LET A = 0
120 GOTO 50
...
```

See TCP Server on page 489 or UDP Server on page 491.

Comments

If there is no error, this line is ignored.

This is a program command that is preceded by a line number.

Applicator Functions

The printer applicator port option can be controlled in part or completely by ZBI 2. When ZBI takes control of a pin, the printer's built-in applicator functionality will not have access to that pin. This function will allow the printer to perform some of the functionality that a programmable logic controller (PLC) could.

AUXPORT_STEALPIN Takes control of a pin and allows ZBI to perform other actions on the pin.

AUXPORT_SETPIN Sets the output level on an applicator pin.

AUXPORT_GETPIN Retrieves the state of the applicator pin.

AUXPORT_RELEASEPIN Returns a pin controlled by ZBI to normal printer operation.

AUXPORT_STEALPIN

This function will take control of a pin and allow ZBI to perform other actions on the pin.

Format AUXPORT_STEALPIN(x)

Parameters x = perform action on this applicator port pin.

Returns This function returns -1 upon failure and 0 upon success.

Example This is an example of the `AUXPORT_STEALPIN` command:

```
1 REM Demo applicator to show control of applicator pins
1 REM on the printer
1 REM The application is to create a light pole with an
1 REM external feed button
AUTONUM 1,1
LET RED = 9
LET YELLOW = 10
LET GREEN = 11
LET BUTTON = 4
LET FEED_KEY = 3
LET TMP = AUXPORT_STEALPIN(RED)
LET TMP = AUXPORT_STEALPIN(YELLOW)
LET TMP = AUXPORT_STEALPIN(GREEN)
LET TMP = AUXPORT_STEALPIN(BUTTON)
DO WHILE 1 = 1
SLEEP 1
IF ISERROR = 1 THEN
LET TMP = AUXPORT_SETPIN(RED,1)
LET TMP = AUXPORT_SETPIN(YELLOW,0)
LET TMP = AUXPORT_SETPIN(GREEN,0)
ELSE IF ISWARNING = 1 THEN
LET TMP = AUXPORT_SETPIN(RED,0)
LET TMP = AUXPORT_SETPIN(YELLOW,1)
LET TMP = AUXPORT_SETPIN(GREEN,0)
ELSE
LET TMP = AUXPORT_SETPIN(RED,0)
LET TMP = AUXPORT_SETPIN(YELLOW,0)
LET TMP = AUXPORT_SETPIN(GREEN,1)
END IF
IF AUXPORT_GETPIN(BUTTON) = 1 THEN
LET A = TRIGGEREVENT(FEED_KEY)
END IF
LOOP
```

Comments If this pin is not controlled via ZBI (power pin), this function will return -1.

AUXPORT_SETPIN

This function sets the output level on an applicator pin.

Format AUXPORT_SETPIN**(X,Y)**

Parameters

X = perform action on this applicator port pin.

Y = The value to set on the pin (1 = high, 0 = low).

Returns This function returns -1 upon failure and 0 upon success.

Example See AUXPORT_STEALPIN on page 518.

Comments If this pin is not controlled via ZBI (power pin), this function will return -1. See AUXPORT_STEALPIN on page 518.

AUXPORT_GETPIN

Description This function will retrieve the state of the applicator pin.

Format AUXPORT_GETPIN**(X)**

Parameters **X** = perform action on this applicator port pin.

Returns This function returns 1 if pin is in high state, 0 in low state, and -1 upon failure.

Example See AUXPORT_STEALPIN on page 518.

Comments If this pin is not controlled via ZBI (power pin), this function will return -1. See AUXPORT_STEALPIN on page 518.

AUXPORT_RELEASEPIN

Description This function returns a pin controlled by ZBI to normal printer operation.

Format AUXPORT_RELEASEPIN**(X)**

Parameters **X** = perform action on this applicator port pin.

Returns This function returns -1 upon failure and 0 upon success.

Example This is an example of the `AUXPORT_RELEASEPIN` command:

```
90 LET TMP = AUXPORT_RELEASEPIN(X)
```

Comments If this pin is not controlled via ZBI (power pin), this function will return -1. See AUXPORT_STEALPIN on page 518.

String Functions

This section identifies how to handle string manipulation. Here is a quick list of these commands:

LCASE$ Converts a string to all lowercase characters.

CHR$ Takes a value between 0 and 255 and puts that value into a string.

LTRIM$ Removes leading spaces from a string.

REPEAT$ Creates multiple copies of a string combined into a new string.

RTRIM$ Returns a string with trailing spaces removed

SPLIT Splits a string into sub-strings

SPLITCOUNT Returns the number of sub-strings that would be returned by the SPLIT function.

UCASE$ Converts a string to all uppercase characters

EXTRACT$ Searches for a string based on a starting and ending string.

ORD Returns the ASCII value of the first character of string `A$`.

POS Returns the location of the first occurrence of a search string in the target string.

LEN Returns the length of a string.

LCASE$

This function will convert a string to all lowercase characters.

Format **`LCASE$ (A$)`**

Parameters **`(A$)`** = the string that will be converted

Returns The characters in **`A$`** converted to lowercase.

Example This is an example of how to use the **`LCASE$`** command.

```
10 LET B$=LCASE$ ("Hello World")
20 PRINT B$
RUN
hello world
```

Comments This will only work on non-accented Latin characters, A-Z.

CHR$

This function takes a value between 0 and 255 and puts that value into a string.

Format `CHR$(VAL)`

Parameters `(VAL)`= The numeric value of the string character.

Returns A single character string containing the value entered.

Example This is an example of how to use the `CHR$` command to easily put control characters into strings:

```
10 LET NULL$=CHR$(0)
20 LET STX$=CHR$(2)
30 LET ETX$=CHR$(3)
40 LET EOT$=CHR$(4)
```

Comments None

LTRIM$

This function removes leading spaces from a string.

Format `LTRIM$(A$)`

Parameters `(A$)` = the string to convert.

Returns The string in `A$` with no spaces.

Example This is an example of how to use the `LTRIM$(A$)` command:

```
10 LET A$=" Hello"
20 PRINT LTRIM$(A$)
RUN
Hello
```

Comments None

REPEAT$

This function creates multiple copies of a string combined into a new string.

Format **REPEAT$(A$,M)**

Parameters

A$ = the base string to duplicate

M = the number of times to duplicate **A$**

Returns A string containing **M** copies of **A$**. **Note:** When M=0, an empty string is returned.

Example This is an example of how to use the REPEAT$(A$,M) command:

```
10 PRINT REPEAT$("Hello",3)
RUN
HelloHelloHello
```

Comments None

RTRIM$

This function returns a string with trailing spaces removed.

Format **RTRIM$(A$)**

Parameters **(A$)** = the base string

Returns **A$** with trailing spaces removed.

Example This is an example of how to use the RTRIM$(A$) command:

```
10 LET A$="Hello "
20 LET B$="World"
30 PRINT A$ & B$
40 PRINT RTRIM$(A$)& B$
RUN
Hello World
HelloWorld
```

Comments None

SPLIT

Description This function allows a string to be split into sub-strings

Format

```
SPLIT(DEST$,SOURCE$,DELIMITER$)
SPLIT(DEST$,SOURCE$,DELIMITER$,MAXCOUNT)
```

Parameters

DEST$ = the array to populate with the sub-strings created by the split

SOURCE$ = the string that will be searched for the provided delimiter

DELIMITER$ = the delimiter string (may be more than one character) to search for

MAXCOUNT = the maximum number of sub-strings the string should be split into. A negative value will return every sub-string created by the split. A value of zero will return empty strings in the array. If not specified, the limit will be the maximum size of the array.

Returns The number of sub-strings placed into the DEST$ array. If the number of sub-strings is less than the size of DEST$, the remaining elements of the array will be set to empty strings.

Example This is an example of how to use the `SPLIT` command:

```
1 REM Example - This example show how the SPLIT and SPLITCOUNT
1 REM commands can be
1 REM used to merge a comma separated variable string(CSV)
1 REM into a stored format
AUTONUM 1,1
SLEEP 10
DECLARE STRING TESTDATA$(5)
REM data format = <Format Name>,<VAR 1>,<VAR 2>,...,<VAR N>
LET TESTDATA$(1) = "E:PRICETAG.ZPL,FRED'S OATS,$1.25,C:126789:325,123456789"
LET TESTDATA$(2) = "E:PRICETAG.ZPL,FRED'S OATS,$2.25,C:126789:325,123456789"
LET TESTDATA$(3) = "E:PRICETAG.ZPL,FRED'S OATS,$3.25,C:126789:325,123456789"
LET TESTDATA$(4) = "E:PRICETAG.ZPL,FRED'S OATS,$4.25,C:123489:325,123456789"
LET TESTDATA$(5) = "E:PRICETAG.ZPL,FRED'S OATS,$5.25,C:123459:325,123456789"
LET ZPLPORT = 2
OPEN #ZPLPORT: NAME "ZPL"
FOR T = 1 TO 5
LET DATA$ = TESTDATA$(T)
GOSUB CSVPRINTER
NEXT T
END
REM ******* Subroutine CSVPRINTER, expects DATA$ and ZPLPORT ***************
SUB CSVPRINTER
LET CNT = SPLITCOUNT(DATA$, ",")
DECLARE STRING SPLITSTRING$(CNT)
ON ERROR GOTO RECOVERY
LET CNT = SPLIT(SPLITSTRING$,DATA$,",")
PRINT #ZPLPORT: "^XA^XF";SPLITSTRING$(1);"^FS"
IF CNT >= 2 THEN
FOR I = 2 TO CNT
PRINT #ZPLPORT: "^FN";I-1;"^FD";SPLITSTRING$(I);"^FS"
NEXT I
END IF
PRINT #ZPLPORT: "^XZ"
SUB RECOVERY
RETURN
```

Example This is an example of how to use the `SPLIT` command:

```
1 REM Example - Shows how the SPLIT and SPLITCOUNT commands can be used to
1 REM merge a comma separated variable string(CSV) into a stored format
AUTONUM 1,1
SLEEP 10
DECLARE STRING TESTDATA$(5)
REM data format = <Format Name>,<VAR 1>,<VAR 2>,...,<VAR N>
LET F$="E:PRICETAG.ZPL"
LET TESTDATA$(1) = F$&",FRED'S ROLLED OATS,$1.25,C:123456789:325,123456789"
LET TESTDATA$(2) = F$&",FRED'S ROLLED OATS,$2.25,C:123456789:325,123456789"
LET TESTDATA$(3) = F$&",FRED'S ROLLED OATS,$3.25,C:123456789:325,123456789"
LET TESTDATA$(4) = F$&",FRED'S ROLLED OATS,$4.25,C:123456789:325,123456789"
LET TESTDATA$(5) = F$&",FRED'S ROLLED OATS,$5.25,C:123456789:325,123456789"
LET ZPLPORT = 2
OPEN #ZPLPORT: NAME "ZPL"
FOR T = 1 TO 5
LET DATA$ = TESTDATA$(T)
GOSUB CSVPRINTER
NEXT T
END
REM ******* Subroutine CSVPRINTER, expects DATA$ and ZPLPORT *****************
SUB CSVPRINTER
LET CNT = SPLITCOUNT(DATA$, ",")
DECLARE STRING SPLITSTRING$(CNT)
ON ERROR GOTO RECOVERY
LET CNT = SPLIT(SPLITSTRING$,DATA$,",")
PRINT #ZPLPORT: "^XA^XF";SPLITSTRING$(1);"^FS"
IF CNT >= 2 THEN
FOR I = 2 TO CNT
PRINT #ZPLPORT: "^FN";I-1;"^FD";SPLITSTRING$(I);"^FS"
NEXT I
END IF
PRINT #ZPLPORT: "^XZ"
SUB RECOVERY
RETURN
```

Comments If the delimiter is an empty string, or does not appear in the **`SOURCE$`** string, the first entry of the array will be the source string and all other elements will be empty strings.

When the **`SPLIT`** function encounters a delimiter at the beginning or end of the source string, or two delimiters in a row, it populates the corresponding array element with an empty string.

If **`MAXCOUNT`** is larger than the number of returned sub-strings (N), the last **`MAXCOUNT`** - N array elements will be empty strings. If **`MAXCOUNT`** is larger than the destination array or is negative, the size of the array will be used as the **`MAXCOUNT`**. Therefore, the smallest value among the value of **`MAXCOUNT`**, the size of the return array, or the number of sub-strings found determines the maximum number of sub-strings that will be returned.

If **`MAXCOUNT`** is less than the number of delimiters in a string the last string in the array will hold the end of the string starting from where the last delimiter was found. For example, if **`SOURCE$`** = "one,two,three,four,five", **`DELIMITER$`** = ",", and **`MAXCOUNT`** = 2, the output would be two strings: "one" and "two,three,four,five".

If a two dimensional array is provided for **`DEST$`**, the array will be filled linearly. For example, an array that is 2 x 3 (for example, **`DECLARE STRING MYARRAY$(2,3)`**) will be filled from (0,0), then (0,1) up to (2,3).

SPLITCOUNT

Description This function returns the number of sub-strings that would be returned by the SPLIT function.

Format **SPLITCOUNT(SOURCE$, DELIMITER$)**

Parameters

SOURCE$ = the string that will be searched for the provided delimiter.

DELIMITER$ =5

Returns The number of sub-strings that would be returned by the **SPLITCOUNT** function.

Example This function shows how to determine the number of sub-strings that the **SPLITCOUNT** command would produce

```
10 LET CNT = SPLITCOUNT("ONE,,,FOUR,FIVE,,SEVEN,", ",")
20 PRINT "Number of sub-strings returned is", STR$(CNT)
RUN
Number of sub-strings returned is 8
```

Comments None

UCASE$

This function converts a string to all uppercase characters.

Format **UCASE$(A$)**

Parameters **A$** = the base string to convert

Returns **A$** converted to uppercase.

Example This is an example of how to use the `UCASE$(A$)` command:

```
10 LET A$="Zebra Technologies"
20 PRINT UCASE$(A$)
RUN
ZEBRA TECHNOLOGIES
```

Example This is an example of how to capitalize a line.

```
10 LET A$="The Cow jUmped Over THE Moon."
20 LET A$=LCASE$(A$)
30 LET A$(1:1)=UCASE$(A$(1:1))
40 PRINT A$
RUN
The cow jumped over the moon.
```

Comments This will only convert non-accented Latin characters, a-z.

EXTRACT$

This function searches for a string based on a starting and ending string. When these two strings are found, the string between them is returned.

IMPORTANT: If the `EXTRACT$` command encounters a carriage return line feed before encountering the beginning character or the ending character, it returns null.

Format

`EXTRACT$ (CHANNEL, START$, STOP$)`

`EXTRACT$ (A$, START$, STOP$)`

Parameters

`CHANNEL` = extracts data from this channel

`A$` = the source string

`START$` = Once this string is found, the extract pulls characters immediately following.

`STOP$` = the extraction stops when this string is found

Example This example shows how to extract the word Technologies from this string: `Zebra,Technologies,Corporation.`

This is what the program looks like to accomplish this:

```
10 LET A$ = "Zebra,Technologies,Corporation,"
20 LET DATA$ = EXTRACT$(A$,",",",",")
```

Example This example shows how the EXTRACT$ command works from an open port:

```
10 OPEN #1: NAME "SER"
20 LET DATA$ = EXTRACT$(1,",",",",")
```

Notice how the quotes are used to show a literal character, in this case a comma.

Example This example shows how the start and stop points are variable; a variable name is used instead of the literal:

```
10 LET B$ = ","
20 LET A$ = "Zebra,Technologies,Corporation"
30 LET DATA$ = EXTRACT$(A$,B$,B$)
40 PRINT DATA$
RUN
Technologies
```

Example This example shows how an empty string can be used to extract from the start of the input string to the end string:

```
10 LET IN$ = "BLAH BLAH <END>"
20 LET B$ = EXTRACT$(IN$, "", "<END>")
30 PRINT B$
RUN
BLAH BLAH
```

Example This example will use an empty string to extract to the end of a line:

```
10 LET IN$ = "BLAH <START> THE DATA"
20 LET B$ = EXTRACT$(IN$, "<START>", "")
30 PRINT B$
RUN
THE DATA
```

Comments EXTRACT$ reads in and discards data until the start criteria is met. Then, all data is returned up to the stop criteria.

ORD

This function returns the ASCII value of the first character of string `A$`.

Format **`ORD(A$)`**

Parameters **`A$`** = Input string: only the first character will be used.

Returns The ASCII value of the first character.

Example This is an example of how to use the `ORD(A$)` command:

```
10 LET A$="ABC"
20 PRINT ORD(A$)
RUN
65
```

Comments None

POS

This function returns the location of the first occurrence of a search string in the target string. It can be assigned an index.

Format

POS(A$,B$)

POS(A$,B$,M)

Parameters

A$ = the target string to search

B$** = the search string to find in **A$

M = The index to start looking for **B$**. If omitted, the search will start at the beginning of the string. **M** must be greater than zero.

Returns The location of the string. If the string is not found, this will return 0.

Example This is an example of how to use the `POS` command:

```
10 LET A$="Hello World"
20 LET B$="o"
30 PRINT POS(A$,B$)
40 PRINT POS(A$,B$,1)
50 PRINT POS(A$,B$,6)
RUN
5
5
8
```

Comments None

LEN

This function returns the length of a string.

Format **LEN(A$)**

Parameters **A$** = the target string from which to determine the length.

Returns The length of the string.

Example This example identifies the length of a string. Hello World is 11 characters, as follows:

```
10 LET A$="Hello World"
20 PRINT LEN(A$)
RUN
11
```

Comments None

Math Functions

This section identifies how to handle mathematical calculations. Here is a quick list of these commands:

STR$ Converts a number to a string.

MAX Returns the greater value between two numbers.

MIN Returns the smaller value of two numbers.

MAXNUM returns the largest number permitted by this machine.

MOD Computes the remainder from division.

VAL Evaluates the number represented by a string.

INTTOHEX$ Takes a numeric value and converts it into a hexadecimal string.

HEXTOINT Converts hexadecimal strings to integers.

STR$

This function converts a number to a string.

Format `STR$(X)`

Parameters **X** = the number to convert to a string

Returns A string representing **X**.

Example This is an example of how to use the `STR$(X)` command:

```
10 LET A=53
20 PRINT STR$(A)
RUN
53
```

Comments None

MAX

This function returns the greater value between two numbers.

Format `MAX(X,Y)`

Parameters

X = the first number to compare

Y = the second number to compare

Returns The greater of **X** or **Y**.

Example This is an example of how to use the `MAX(X,Y)` command:

```
10 LET A=-2
20 LET B=1
30 PRINT MAX(A,B)
RUN
1
```

Comments None

MIN

This function returns the smaller value of two numbers.

Format `MIN(X,Y)`

Parameters

X = the first number to compare

Y = the second number to compare

Returns The smaller of **X** or **Y**.

Example This is an example of how to use the `MIN(X,Y)` command:

```
10 LET A=-2
20 LET B=0
30 PRINT MIN(A,B)
RUN
-2
```

Comments None

MAXNUM

This function returns the largest number permitted by this machine: 2,147,483,647.

Format `MAXNUM`

Parameters N/A

Returns The largest number that the NUMERIC type can handle (2,147,483,647).

Example This is an example of how to use the `MAXNUM` command:

```
10 PRINT MAXNUM
RUN
2147483647
```

Comments None

MOD

This function computes the remainder from division. (This is known as the modulus.)

Format `MOD(X,Y)`

Parameters

`X` = the value to be modulated (numerator).

`Y` = the base number or divisor (denominator).

Returns The remainder of the division (X/Y).

Example This is an example of how to use the `MOD(X,Y)` command:

```
10 PRINT MOD(25,10)
20 PRINT MOD(2,1)
30 PRINT MOD(3,2)
40 PRINT MOD(9,2)
50 PRINT MOD(-2,9)
60 PRINT MOD(2,0)
RUN
5
0
1
1
-2
ERROR OCCURRED ON LINE 60:DIVIDE BY ZERO
```

Comments None

VAL

This function evaluates the number represented by a string.

Format `VAL(A$)`

Parameters `A$` = This is the input string to pull the number from. Non-numbers are ignored.

Returns The numeric representation of the string.

Example This is an example of how to use the `VAL(A$)` command:

```
10 LET A$="123"
20 LET C=VAL(A$)
30 PRINT C
RUN
123

PRINT VAL("321A123")
321123
```

Comments None

INTTOHEX$

Description This function will take a numeric value and convert it into a hexadecimal string. The range of values for integers is:

-2,147,483,648 to +2,147,483,647

Format **`INTTOHEX$(A)`**

Parameters **`A`** = The numeric value to convert.

Returns A string representing the integer in hex.

Example These print statements show the output of the **`INTTOHEX$`** function given different values.

```
PRINT INTTOHEX$(1)
1

PRINT INTTOHEX$(10)
A

PRINT INTTOHEX$(16)
10

PRINT INTTOHEX$(20)
14

PRINT INTTOHEX$(30)
1E

PRINT INTTOHEX$(100)
64

PRINT INTTOHEX$(123124)
1EOF4

PRINT INTTOHEX$(-5)
0

PRINT INTTOHEX$(-99)
0
```

Comments Negative values will be returned as 0.

HEXTOINT

This function will convert hexadecimal strings to integers.

Format HEXTOINT**(A$)**

Parameters **A$** = The hex string to convert.

Returns A integer string computed from the hexadecimal string.

Example These print statements show the output of the **INTTOHEX** function given different values.

```
PRINT HEXTOINT("0")
0

PRINT HEXTOINT("A")
10

PRINT HEXTOINT("a")
10

PRINT HEXTOINT("1A")
26

PRINT HEXTOINT("10")
16

PRINT HEXTOINT("AaAa")
43690

PRINT HEXTOINT("AAAA")
43690

PRINT HEXTOINT("-1")
0

PRINT HEXTOINT("-A")
0
```

Comments Negative values will be returned as 0.

Array Functions

This section describes the functions to search, resize, and query arrays.

REDIM Changes the size of an array.

INSERTROW Inserts a new row into an existing array.

DELROW Deletes a new row from an existing array

ROWSIZE Returns the number of rows in an array.

COLUMNSIZE Returns the number of columns in an array.

FIND Searches a string array for an occurrence of a sub-string.

REDIM

This command will change the dimensions of an array.

Format

```
REDIM <ARRAYNAME>(<SIZE>)
REDIM <ARRAYNAME>(<ROWS>,<COLUMNS>)
REDIM <ARRAYNAME$>(<SIZE>)
REDIM <ARRAYNAME$>(<ROWS>,<COLUMNS>)
```

Parameters

`<SIZE>` = new number of entries in a single dimension array.

`<ROWS>` = new number of rows in a two dimensional array.

`<COLUMNS>` = new number of columns in a two dimensional array.

Example This example shows how to change a one dimensional numeric array.

```
10  DECLARE NUMERIC SCORES(3)
20  LET SCORES(1) = 85
30  LET SCORES(2) = 92
40  LET SCORES(3) = 98
50  REDIM SCORES(2) ! Discard the last one
```

Example This example shows how to change a two dimensional string array.

```
10  DECLARE STRING NAMEAGES$(3,2)
20  LET NAMEAGES$(1,1) = "Abraham"
30  LET NAMEAGES$(1,2) = "Lincoln"
40  LET NAMEAGES$(2,1) = "Dwight"
50  LET NAMEAGES$(2,2) = "Eisenhower"
60  LET NAMEAGES$(3,1) = "Theodore"
70  LET NAMEAGES$(3,2) = "Roosevelt"
80  REDIM NAMEAGES$(5,2) ! Make room for more
```

Comments The **REDIM** must have the same number of dimensions as the original declaration of the array.

- If the array has two dimensions, the second array bound cannot change. It must have the same value as the original declaration.
- If **REDIM** makes an array smaller, elements (or rows, for a two dimensional array) at the end of the array are discarded.
- If **REDIM** makes an array larger, elements (or rows) are added at the end of the array, and initialized as they would be with a **DECLARE**.

This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

INSERTROW

This command will insert a new row into an existing array.

Format `INSERTROW (<ARRAYNAME>, <INDEX>)`

Parameters

<ARRAYNAME> = array where the row will be inserted

<INDEX> = index of the row in the array that the new row will be inserted before

Example This example shows how to insert a row into the middle of an array.

```
10  DECLARE NUMERIC SCORES(3)
20  LET SCORES(1) = 85
30  LET SCORES(2) = 92
40  LET SCORES(3) = 98
50  INSERTROW(SCORES, 2)
60  LET SCORES(2) = 100
```

Example This example shows how to add a row into the end of an array.

```
10  DECLARE NUMERIC SCORES(3)
20  LET SCORES(1) = 85
30  LET SCORES(2) = 92
40  LET SCORES(3) = 98
50  INSERTROW(SCORES, 4)
60  LET SCORES(4) = 100
```

Comments Inserting a row increases the size of the array by one row, and moves all the rows from **INDEX** to the end of the array up one row, leaving an empty row at position **INDEX**.

INDEX cannot be any larger the number of rows in the array plus one. If the number of rows plus one is provided, the new row will be added to the end of the array.

This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

DELROW

This command will delete a row from an existing array.

Format `DELROW (<ARRAYNAME>, <INDEX>)`

Parameters

<ARRAYNAME> = the array where the row will be deleted

<INDEX> = index of the row to delete from the array

Example This example shows how to delete a row from the middle of an array.

```
10  DECLARE NUMERIC SCORES(5)
20  LET SCORES(1) = 85
30  LET SCORES(2) = 92
40  LET SCORES(3) = 98
50  LET SCORES(4) = 45
60  LET SCORES(5) = 100
70  DELROW(SCORES, 4) ! Remove the low score
```

Comments This decreases the size of A by one row, and moves all the rows from INDEX to the end of the array down by one, overwriting the row at position INDEX.

INDEX cannot be any larger the number of rows in the array.

If the array only has one row, that row may not be deleted.

This can be an interactive command that takes effect as soon as it is received by the printer, or a program command that is preceded by a line number.

ROWSIZE

This function will return the number of rows in an array.

Format

```
ROWSIZE(A)
ROWSIZE(A$)
```

Parameters

`A` = integer array to query for the number of rows.

`A$` = string array to query for the number of rows.

Returns Returns a 0 if the variable is not an array. Returns the number of elements in the array if the array has only one dimension. Returns the size of the first dimension if the array has two dimensions.

Example This example shows how to determine the number of elements in a one dimensional string array.

```
10 DECLARE STRING NAMES$(3)
20 LET NAMES$(1) = "Fred"
30 LET NAMES$(2) = "Wilma"
40 LET NAMES$(3) = "Barney"
50 REDIM NAMES$(4) ! Make room for Betty
60 LET NAMES$(4) = "Betty"
70 LET NUMOFNAMES = ROWSIZE(NAMES$)
80 PRINT NUMOFNAMES
```

Example This example shows how to determine the number of rows in a two dimensional numeric array.

```
10 DECLARE NUMERIC SQROFTWOLOOKUP(3,2)
20 LET SQROFTWOLOOKUP (1,1) = 1
30 LET SQROFTWOLOOKUP (1,2) = 2
40 LET SQROFTWOLOOKUP (2,1) = 2
50 LET SQROFTWOLOOKUP (2,2) = 4
60 LET SQROFTWOLOOKUP (3,1) = 3
70 LET SQROFTWOLOOKUP (3,2) = 8
80 LET NUMOFSQRS = ROWSIZE(SQROFTWOLOOKUP)
90 PRINT NUMOFSQRS
```

COLUMNSIZE

This function will return the number of columns in an array.

Format

```
COLUMNSIZE(A)
COLUMNSIZE(A$)
```

Parameters

`A` = integer array to query for the number of columns.

`A$` = string array to query for the number of columns.

Returns A 0 if the variable is not an array. Returns 1 if the array has only one dimension. Returns the size of the second dimension if the array has two dimensions.

Example This example shows how to determine the number of elements in a one dimensional string array.

```
10 DECLARE STRING NAMES$(3)
20 LET NAMES$(1) = "Fred"
30 LET NAMES$(2) = "Wilma"
40 LET NAMES$(3) = "Barney"
50 REDIM NAMES$(4) ! Make room for Betty
60 LET NAMES$(4) = "Betty"
70 LET NUMOFCOLS = COLUMNSIZE(NAMES$)
80 PRINT NUMOFCOLS
```

Example This example shows how to determine the number of columns in a two dimensional numeric array.

```
10 DECLARE NUMERIC SQROFTWOLOOKUP(3,2)
20 LET SQROFTWOLOOKUP (1,1) = 1
30 LET SQROFTWOLOOKUP (1,2) = 2
40 LET SQROFTWOLOOKUP (2,1) = 2
50 LET SQROFTWOLOOKUP (2,2) = 4
60 LET SQROFTWOLOOKUP (3,1) = 3
70 LET SQROFTWOLOOKUP (3,2) = 8
80 LET COLCNT = COLUMNSIZE(SQROFTWOLOOKUP)
90 PRINT COLCNT
```

FIND

This function will find an element of a string array that contains an identified search string.

Format

```
FIND(A$, B$)
FIND(A$, B$, START)
FIND(A$, COLUMN, B$)
FIND(A$, COLUMN, B$, START)
```

Parameters

`A$` = string array to search for `B$`.

`B$` = string to search for within `A$`.

`START` = index within a single dimensional array, or row for a two dimensional array, to start the search.

`COLUMN` = column to isolate search to in a two dimensional array. This must be supplied if `A$` is a two dimensional array.

Returns Returns a 0 if `B$` is not found or if there was an error. Otherwise, returns the index that contains the first occurrence of the string `B$` (the element index for one dimensional arrays, the row for two dimensional arrays).

Example This example shows how to find a string in a one dimensional array.

```
10 DECLARE STRING NAMES$(4)
20 LET NAMES$(1) = "Fred"
30 LET NAMES$(2) = "Wilma"
40 LET NAMES$(3) = "Barney"
50 LET NAMES$(4) = "Betty"
60 LET BARNEYIX = FIND(NAMES$, "Bar")
70 PRINT "Found Barney in element "; STR$(BARNEYIX)
```

Example This example shows how to find a string that occurs more than once in a two dimensional array.

```
10  DECLARE STRING CLOTHING$(5,2)
20  LET TYPECOL     = 1
30  LET MATERIALCOL = 2
40  LET CLOTHING$(1,1) = "Gloves"
50  LET CLOTHING$(1,2) = "Knit"
60  LET CLOTHING$(2,1) = "Pants"
70  LET CLOTHING$(2,2) = "Cotton"
80  LET CLOTHING$(3,1) = "Gloves"
90  LET CLOTHING$(3,2) = "Leather"
100 LET CLOTHING$(4,2) = "Shirts"
110 LET CLOTHING$(4,2) = "Polyester"
120 LET CLOTHING$(5,2) = "Pants"
130 LET CLOTHING$(5,2) = "Denim"
140 LET GLOVEIX = 1
150 DO
160 LET GLOVEIX = FIND(CLOTHING$, TYPECOL, "Gloves", GLOVEIX)
170 IF NOT GLOVEIX = 0 THEN
180 PRINT CLOTHING$(GLOVEIX, MATERIALCOL), "gloves are available"
190 LET GLOVEIX = GLOVEIX + 1
200 END IF
210 LOOP WHILE NOT GLOVEIX = 0
```

Comments **COLUMN** must be greater than 0.

If START is given, it must be greater than 0.

FIND will match the first occurrence of B$, even if it is a substring of a string within the A$ array. For example, “Coat” will be found in both locations 1 and 4.

```
5 DECLARE STRING A$(5)
10 LET A$(1) = “Over Coat”
20 LET A$(2) = “Hat”
30 LET A$(3) = “Jacket”
40 LET A$(4) = “Coat”
50 LET A$(5) = “Boots”
```

If an exact match is needed, **FIND** should be called until 0 is returned or the item is found and confirmed. To confirm, check the item against the expected item, it should match exactly. See CSV Program on page 548 for an example showing how to do this.

Time and Date Functions

This section describes the functions to access the real time clock option. Here is a quick list of these commands:

DATE$ Returns the date as a string

TIME$ Returns the current time in a string.

DATE Gets the current date as a number.

TIME Gets the current time as a number.

DATE$

This function returns the date as a string.

Format **DATE$**

Parameters N/A

Returns The current date in string form `YYYYMMDD`. If the Real-Time Clock is not installed, an empty string is returned.

Example This is an example of how to use the `DATE$` command:

```
10 PRINT DATE$
RUN
```

The result, assuming the date is January 1, 2003 is:

```
20030101
```

Example This is another example of the `DATE$` command used with the sub-string operator to get the day of the month:

```
10 LET A$=DATE$(7:8)
20 IF A$ <> DATE$(7:8)
30 LET A$=DATE$(7:8)
40 IF A$="01"
50 PRINT "IT IS THE FIRST OF THE MONTH"
60 END IF
70 END IF
80 SLEEP 100
90 GOTO 20
```

Comments None

TIME$

This function returns the current time in a string.

Format `TIME$`

Parameters N/A

Returns This function returns the time of day in format HH:MM:SS (hours:minutes:seconds). If the Real-Time Clock is not installed, an empty string is returned.

Example This is an example of how to use the `TIME$`command:

```
10 PRINT TIME$
RUN
10:00:00
```

DATE

This function gets the current date as a number.

Format `DATE`

Parameters N/A

Returns This function returns the current date in **YYYYDDD** format, where **YYYY** is the year and **DDD** is the number of days since the beginning of the year. If the Real-Time Clock is not installed, **0** is returned.

Example This example assumes the current date is January 1, 2003:

```
10 PRINT DATE
RUN
2003001
```

TIME

This function gets the current time as a number.

Format `TIME`

Parameters N/A

Returns This function returns the time past midnight (2400h) in seconds. If the Real-Time Clock is not installed, 0 is returned.

Example This is an example of how to use the `TIME` command [assuming the time is one minute past midnight]:

```
10 PRINT TIME
RUN
60
```

Set/Get/Do Interactions

The printer's Set/Get/Do data can be directly accessed via ZBI. For a complete listing of what can be accessed, type the following:

```
! U1 getvar "allcv"
```

Here's a quick list of these commands:

SETVAR Allows the direct setting of printer parameters.

GETVAR$ Retrieves printer parameters.

SETVAR

Description **SETVAR** allows the direct setting of printer parameters.

Format **SETVAR (PARAM$, VALUE$)**

Parameters

PARAM$ = The printer parameter to set.

VALUE$ = the value to set

Returns Parameter dependent.

Example This is an example of the **SETVAR** command:

```
AUTONUM 1,1
LET OUTSTR$ = "Processing"
LET LOOPCTR = 200
LET TIMER5 = 17
LET TMP = REGISTEREVENT(TIMER5, 0, 1000)
DO WHILE LOOPCTR > 0
LET EVT = HANDLEEVENT()
IF EVT = TIMER5 THEN
LET A = SETVAR("device.frontpanel.line2",OUTSTR$)
LET OUTSTR$ = OUTSTR$ & "."
IF LEN(OUTSTR$) >16 THEN
LET OUTSTR$ = "Processing"
END IF
END IF
LET LOOPCTR = LOOPCTR - 1
SLEEP 1
LOOP
LET TMP = UNREGISTEREVENT(TIMER5)
LET A = SETVAR("device.frontpanel.line2","")
END
```

Comments None

GETVAR$

This function retrieves printer parameters.

Format `GETVAR$ (PARAM$)`

Parameters

`PARAM$` = the printer parameter to get.

Returns The value of the parameter. Refer to the SGD commands for specific parameters.

Example **Example:** This is an example of the `GETVAR$` command:

```
AUTONUM 1,1
LET SGDCOUNT = 7
DECLARE STRING SGDQUERY$(2,SGDCOUNT)
LET SGDQUERY$(1,1) = "appl.name"
LET SGDQUERY$(1,2) = "device.printhead.serialnum"
LET SGDQUERY$(1,3) = "internal_wired.ip.addr"
LET SGDQUERY$(1,4) = "internal_wired.ip.netmask"
LET SGDQUERY$(1,5) = "internal_wired.ip.gateway"
LET SGDQUERY$(1,6) = "internal_wired.ip.port"
LET SGDQUERY$(1,7) = "internal_wired.mac_addr"
FOR I = 1 TO SGDCOUNT
LET SGDQUERY$(2,I) = GETVAR$(SGDQUERY$(1,I))
NEXT I
OPEN #1: NAME "ZPL"
PRINT #1: "^XA"
FOR I = 1 TO SGDCOUNT
PRINT #1: "^FO50,";50*I;"^A0N,25,25^FD";SGDQUERY$(1,I);"=";
PRINT #1: SGDQUERY$(2,I);"^FS"
NEXT I
PRINT #1: "^XZ"
```

Comments None

Example Programs

The next section provides example programs of common tasks using ZBI commands. These programs are also available for download at: www.zebra.com/zbi

Array Program

This program prompts a user to enter first a name; when it is entered, it is added to an array of all names entered. The user is them prompted to enter an address, which is then added to an array of all addresses entered. After the user enters a total or five names and addresses, the program uses the arrays to print the entered data on five labels.

Example This is an example of Array

```
1 rem ********************************************************
1 rem Zebra Technologies ZBI Sample Program
1 rem
1 rem Professional programming services are available. Please contact
1 rem ZBI-Experts@zebra.com for more information.
1 rem
1 rem This is an example of using arrays to store and use data within 1 rem ZBI.
1 rem ********************************************************
1 rem close all ports except for the console
1 rem********************************************************
10 for i = 1 to 9 step 1
20    close #i
30    next i
1 rem ********************************************************
1 rem open a port to the print engine
1 rem ********************************************************
40 open #1: name "ZPL"
1 rem ********************************************************
1 rem create string arrays five elements in size to hold names and
1 rem addresses
1 rem ********************************************************
50 declare string name$(5)
60 declare string address$(5)
1 rem ********************************************************
1 rem infinite loop to put name and address data from console into
1 rem arrays
1 rem ********************************************************
70 do
80 for i = 1 to 5 step 1
90    print "PLEASE ENTER THE NAME"
1 rem ********************************************************
1 rem get data from console; input command looks for CRLF
1 rem ********************************************************
100    input name$(i)
1 rem ********************************************************
1 rem if the user inputs end or END, the program will end
1 rem ********************************************************
110    if name$(i) = "END" or name$(i) = "end" then
120       end
130    end if
140    print "PLEASE ENTER THE ADDRESS"
150    input address$(i)
160    if address$(i) = "END" or address$(i) = "end" then
170       end
180    end if
190 next i
200 for index = 1 to 5 step 1 ! For loop To Print data no label
1 rem ********************************************************
1 rem semicolon at the end prints with no CRLF
```

```
1 rem ********************************************************
210    print #1: "^XA^FO30,30^A0N,30,30^FD"&NAME$(INDEX)&"^FS";
1 rem ********************************************************
1 rem ampersand used to concatenate data into strings
1 rem ********************************************************
220    print #1: "^FO30,70^A0N,30,30^FD"&ADDRESS$(INDEX)&"^FS^XZ"
230 next index
240 loop ! loops back To Line 60
250 end
```

CSV Program

The following program will initialize and then execute continuously, repeating the same series of operations; process events, read input from the serial port, write any processed data out to the ZPL port, and then process the data read from the serial port.

The program first loads the CSV database E:PRODUCTS.CSV (in PROGRAMINIT subroutine). Then, data read from the serial port is compared against the first column in the database. If an entry is found in the first column of a row (in FINDITEM subroutine), the data for the respective row is inserted into the ZPL format E:PRICELBL.ZPL and printed on a label.

Example This is an example of a CSV program.

```
1 REM SUBROUTINES BELOW....
2 REM
3 REM ***************************************************************************
4 REM              MAIN LOOP - DO NOT MODIFY
5 REM ***************************************************************************
6 REM
7 GOSUB PROGRAMINIT
8 DO WHILE 1 = 1
9 GOSUB PROCESSEVENTS
10 GOSUB GETINPUT
11 GOSUB WRITEOUTPUT
12 GOSUB PROCESSDATA
13 LOOP
14 REM SUBROUTINES BELOW....
15 REM
16 REM ***************************************************************************
17 REM              Program Init
18 REM ***************************************************************************
19 REM
20 SUB PROGRAMINIT
21 LET INPORT = 1
22 LET OUTPORT = 2
23 LET ENDLINE$ = CHR$ ( 13 ) & CHR$ ( 10 )
24 OPEN # INPORT : NAME "SER"
25 OPEN # OUTPORT : NAME "ZPL"
26 DECLARE STRING DATABASE$ ( 1 , 1 )
27 LET COLUMNCOUNT = CSVLOAD ( DATABASE$ , "E:PRODUCTS.CSV" )
28 LET OUTDATA$ = "TABLE WITH " & STR$ ( COLUMNCOUNT ) & " COLUMNS LOADED" & ENDLINE$
29 RETURN
30 REM
31 REM ***************************************************************************
32 REM              Process Events
33 REM ***************************************************************************
34 REM
35 SUB PROCESSEVENTS
36 RETURN
37 REM
38 REM ***************************************************************************
39 REM              Get Input
40 REM
41 REM Writes All Data from the serial port to the string INDATA$
```

```
42 REM ***************************************************************************
43 REM
44 SUB GETINPUT
45 IF LEN ( INDATA$ ) < 5000 THEN
46 LET INCOUNT = READ ( INPORT , A$ , 1024 )
47 LET INDATA$ = INDATA$ & A$
48 END IF
49 RETURN
50 REM
51 REM ***************************************************************************
52 REM              Write Output
53 REM
54 REM Writes All Data from the string OUTDATA$ to the ZPL Port
55 REM ***************************************************************************
56 REM
57 SUB WRITEOUTPUT
58 LET OUTCOUNT = WRITE ( OUTPORT , OUTDATA$ , LEN ( OUTDATA$ ) )
59 IF OUTCOUNT > 0 THEN
60 LET OUTDATA$ ( 1 : OUTCOUNT ) = ""
61 END IF
62 RETURN
63 REM
64 REM ***************************************************************************
65 REM              Process Data
66 REM
67 REM Parse the data in the string INDATA$ and write output to OUTDATA$
68 REM ***************************************************************************
69 REM
70 SUB PROCESSDATA
71 IF LEN ( OUTDATA$ ) > 1000 THEN
72 RETURN
73 END IF
74 REM REMOVE ALL LINE FEEDS
75 DO
76 LET LOC = POS ( INDATA$ , CHR$ ( 10 ) )
77 LET INDATA$ ( LOC : LOC ) = ""
78 LOOP WHILE LOC > 0
79 REM COMPLETED LINE FEED REMOVAL
80 LET LOC = POS ( INDATA$ , CHR$ ( 13 ) ) ! Line ends with CR
81 IF LOC > 0 THEN
82 LET INLINE$ = INDATA$ ( 1 : LOC - 1 )
83 LET INDATA$ ( 1 : LOC ) = ""
84 GOSUB FINDITEM
85 IF ROW > 0 THEN
86 LET OUTDATA$ = OUTDATA$ & "^XA^XFE:PRICELBL.ZPL^FS" & ENDLINE$
87 LET OUTDATA$ = OUTDATA$ & "^FN1^FD" & DATABASE$ ( ROW , 1 ) & "^FS" & ENDLINE$
88 LET OUTDATA$ = OUTDATA$ & "^FN2^FD" & DATABASE$ ( ROW , 2 ) & "^FS" & ENDLINE$
89 LET OUTDATA$ = OUTDATA$ & "^FN3^FD" & DATABASE$ ( ROW , 3 ) & "^FS^XZ" & ENDLINE$
90 END IF
91 END IF
92 RETURN
93 REM
94 REM ***************************************************************************
95 REM              Find Item
96 REM
97 REM Search the first column of the database for the exact item requested
98 REM ***************************************************************************
99 REM
100 SUB FINDITEM
101 LET ROW = 0
102 LET EXPECTED$ = INLINE$
103 DO
104 LET FOUNDENTRY$ = ""
105 LET ROW = FIND ( DATABASE$ , 1 , EXPECTED$ , ROW + 1 )
106 IF ROW <> 0 THEN
107 LET FOUNDENTRY$ = DATABASE$ ( ROW , 1 )
```

```
108 END IF
109 LOOP WHILE ( ROW <> 0 AND FOUNDENTRY$ <> EXPECTED$ )
110 RETURN
```

DPI Conversion Program

This program converts a ZPL format being sent to the printer on the parallel port to 300 dpi (dots per inch) from 200 dpi (dots per inch). This is done by searching for and extracting ZPL commands with resolution-dependent arguments and scaling the arguments for a 300 dpi printer.

Example This is an example of dpi conversion:

```
1 rem ******************************************************
1 rem Zebra Technologies ZBI Sample Program
1 rem
1 rem Professional programming services are available. Please contact
1 rem ZBI-Experts@zebra.com for more information.
1 rem
1 rem This is an example of converting a printer from 200 dpi (dots
1 rem per inch
1 rem to 300 dpi. This example covers only some of the ZPL commands
1 rem that
1 rem could be affected by converting from 200 to 300 dpi printing.
1 rem ******************************************************
1 rem open the ports for input and output
1 rem ******************************************************
10 close #1
20 close #2
30 open #1 : name "PAR"
40 open #2 : name "ZPL"
1 rem ******************************************************
1 rem create an array with the search parameters
1 rem ******************************************************
50 declare string find$(20)
60 let find$(1) = "^FO"
70 let find$(2) = "^A0"
80 let find$(3) = "^GB"
90 let find$(4) = "^XZ"
100 let find$(5) = "^A@"
110 let find$(6) = "^LL"
120 let find$(7) = "^LH"
130 let find$(8) = "FO"
140 let find$(9) = "A0"
150 let find$(10) = "GB"
160 let find$(11) = "XZ"
170 let find$(12) = "A@"
180 let find$(14) = "LH"
190 let find$(15) = "^BY"
200 let find$(16) = "BY"
210 let find$(17) = "^B3"
220 let find$(18) = "B3"
1 rem ********************************************************
1 rem search for the parameters
1 rem ********************************************************
300 do
310   let in$ = searchto$(1, find$, 2)
1 rem ************************************************************
1 rem once a parameter is found, determine how to handle it
1 rem ************************************************************
320   if in$ = "^FO" or in$ = "FO" then
330     gosub 520
340   else if in$ = "^LH" or in$ = "LH" then
350     gosub 520
360   else if in$ = "^A0" or in$ = "A0" then
```

```
370     gosub 700
380   else if in$ = "^A@" or in$ = "A@" then
390     gosub 700
400   else if in$ = "^GB" or in$ = "GB" then
410     gosub 1100
420   else if in$ = "^LL" then
430     gosub 1300
440   else if in$ = "^BY" or in$ = "BY" then
450     gosub 1400
460   else if in$ = "^B3" or in$ = "B3" then
470     gosub 1600
480   else if in$ = "^XZ" then
490     print #2: in$;
500   end if
510 loop
1 rem ********************************************************
1 rem convert the ^FO and ^LH commands from 200 to 300 dpi
1 rem ********************************************************
520 inbyte #1: a$
530 let a = ord(a$)
540 if a >= 65 then
550   print #2: in$&a$;
560   goto 660
570 end if
580 let x$ = extract$(1, "", ", ")
590 let x2$ = a$&x$
600 let y$ = extract$(1, "", "^")
610 let x = val(x2$)
620 let y = val(y$)
630 let x2 = (x/2)+x
640 let y2 = (y/2)+y
650 print #2: in$; x2; ","; y2; "^";
660 return
1 rem ********************************************************
1 rem convert the ^A0 and ^A@ commands from 200 to 300 dpi
1 rem ********************************************************
700 inbyte #1: a$
710 let a = ord(a$)
720 let b = 0
730 let c = 0
740 if a >= 65 then
750   print #2: in$&a$; ",";
760   let b = 1
770 end if
780 inbyte #1: a$
790 let h$ = extract$(1, "", ",")
800 if in$ = "^A@" or in$ = "A@" then
810   let c = 1
820    let w$ = extract$(1, "", ",")
830   let m$ = extract$(1, "", "^")
840 else
850   let w$ = extract$(1, "", "^")
860 end if
870 let h = val(h$)
880 let w = val(w$)
900 let h2 = (h/2) + h
910 let w2 = (w/2) + w
920 if b = 1 then
930   print #2: h2; ","; w2;
940 else
950   print #2: in$&"N,"; h2; ","; w2;
960 end if
970 if c = 1 then
980   print #2: ","; m$;
990 end if
1000 print #2: "^";
```

```
1010 return
1 rem ************************************************************
1 rem convert the ^GB command from 200 to 300 dpi
1 rem ************************************************************
1020 let w$ = extract$(1, "", ",")
1030 let h$ = extract$(1, "", ",")
1040 let t$ = extract$(1, "", "^")
1050 let h = val(h$)
1060 let w = val(w$)
1070 let t = val(t$)
1080 let h2 = (h/2)+ h
1090 let w2 = (w/2)+ w
1100 let t2 = (t/2)+ t
1110 print #2: in$; w2; ","; h2; ","; t2; "^";
1120 return
1 rem ************************************************************
1 rem convert the ^LL command from 200 to 300 dpi
1 rem ************************************************************
1300 let l$ = extract$(1, "", "^")
1310 let l = VAL(l$)
1320 let l2 = (l/2) + l
1330 print #2: in$; l2; "^";
1340 return
1 rem ************************************************************
1 rem convert the ^BY command from 200 to 300 dpi
1 rem ************************************************************
1400 inbyte #1: a$
1410 let a = ord(a$)
1420 if a >= 48 and a <= 57 then
1460    let x$ = extract$(1, "", ", ")
1470    let x2$ = a$&x$
1480    let x = val(x2$)
1490    let x2 = (x/2) + x
1500       if x2 > 10 then
1510       let x2 = 10
1520       end if
1530    print #2: in$; x2; ",";
1540 else
1550    print #2: in$; a$;
1560 end if
1570 return
1 rem ************************************************************
1 rem convert the ^B3 command from 200 to 300 dpi
1 rem ************************************************************
1600 let o$ = extract$(1, "", ", ")
1610 let e$ = extract$(1, "", ", ")
1620 let h$ = extract$(1, "", ", ")
1630 let h = val(h$)
1640 let h2 = (h/2) + h
1650 print #2: in$; o$; ","; e$; ","; h2; ",";
1660 return
```

Email Program

This program sends a simple email message to user@domain.com, assuming a valid email server is set up by identifying the SMTP server on the print server. In order to write email via ZBI, the port written to must be named "EML".

Example This is an example of email

```
1 rem ************************************************
1 rem Zebra Technologies ZBI Sample Program
1 rem
1 rem Professional programming services are available. Please contact
1 rem ZBI-Experts@zebra.com for more information.
1 rem
1 rem This is an example of connecting to an email server to send
1 rem email.
1 rem ************************************************
1 rem EOT$ is the special character used to denote end of transmission
1 rem ************************************************
5 let EOT$ = chr$(4)
1 rem ************************************************
1 rem Open a connection to the email port; if there is an error, try
1 rem again
1 rem ************************************************
10 open #1: name "EML"
15 on error goto 10
1 rem ************************************************
1 rem Specify address to send message to, signal end of recipients
1 rem with EOT$
1 rem Note: To send to multiple addressees, separate addressees with
1 rem a space
1 rem ************************************************
20 print #1: "user@domain.com";EOT$;
1 rem ************************************************
1 rem Fill in the message information
1 rem ************************************************
30 print #1: "From: Sample User"
40 print #1: "To: Recipient"
50 print #1: "Subject: This is a test"
60 print #1: ""
70 print #1: "Hello!"
80 print #1: i
1 rem ************************************************
1 rem Terminate message
1 rem ************************************************
90 print #1: "";EOT$
1 rem ************************************************
1 rem Close the port, since each open port is only good for sending
1 rem one message
1 rem ************************************************
100 close #1
110 sleep 2
120 let i = i + 1
130 goto 10
```

Extraction 1 Program

This program finds and stores data of interest, which in this case is found in a format after the string "DATA = ". The extract command is used to get the data from the input stream, and it is inserted into a simple ZPL format to be printed.

Example This is an example of Extraction 1.

```
1 rem ********************************************************
1 rem Zebra Technologies ZBI Sample Program
1 rem
1 rem Professional programming services are available. Please contact
1 rem ZBI-Experts@zebra.com for more information.
1 rem
1 rem This is an example of using ZBI for data extraction.
1 rem There are two methods for doing extraction; this example shows
1 rem data extraction using a string.
1 rem
1 rem The data to extract is as follows:
1 rem START
1 rem DATA = "hello":
1 rem DATA = "goodbye":
1 rem END
1 rem ********************************************************
1 rem close ports except console, open channels to parallel and serial
1 rem ports
1 rem ********************************************************
05 for i = 1 to 9 step 1
10    close #i
20 next i
30 open #1: name "PAR"
40 open #2: name "ZPL"
1 rem ********************************************************
1 rem create string array to hold data
1 rem ********************************************************
50 declare string format$(3)
60 let format$(1) = "START"
70 let format$(2) = "END"
80 let format$(3) = "DATA"
1 rem ********************************************************
1 rem main program; look for "START" keyword, if found print ^XA to ZPL port
1 rem ********************************************************
90 do
100    let begin$ = searchto$(1,format$,2)
110    if begin$ = "START" then
120    print #2: "^XA";
1 rem ********************************************************
1 rem if "DATA" keyword is found, get two data strings
1 rem ********************************************************
130    else if begin$ = "DATA" then
140       input #1: data_string1$
150       input #1: data_string2$
1 rem ********************************************************
1 rem get data from between quotes and print to ZPL port with formatting
1 rem ********************************************************
160       let extracted_data1$ = extract$(data_string1$,"""","""")
170       let extracted_data2$ = extract$(data_string2$,"""","""")
180       print #2:"^FO30,30^A0N,30,30^FD"&extracted_data1$&"^FS";
190       print #2:"^FO30,70^A0N,30,30^FD"&extracted_data2$&"^FS";
200    else if begin$ = "END" then
210       print #2: "^XZ                    "
220    end if
230 loop
```

Extraction 2 Program

This program finds and stores data of interest, which in this case is found in a format after the string "DATA = ". The input command is used to get the data from the input stream, and it is inserted into a simple ZPL format to be printed.

Example This is an example of Extraction 2.

```
1 rem*******************************************************
1 rem Zebra Technologies ZBI Sample Program
1 rem
1 rem Professional programming services are available. Please contact
1 rem ZBI-Experts@zebra.com for more information.
1 rem
1 rem This is an example of using ZBI for data extraction.
1 rem There are two methods for doing extraction; this example shows
1 rem data extraction from the port directly.
1 rem
1 rem The data to extract is as follows:
1 rem START
1 rem DATA = "hello":
1 rem DATA = "goodbye":
1 rem END
1 rem*******************************************************
1 rem close ports except console, open channels to parallel and serial ports
1 rem*******************************************************
05 for i = 1 to 9 step 1
10    close #i
20 next i
30 open #1: name "PAR"
40 open #2: name "ZPL"
1 rem*******************************************************
1 rem create string array to hold data
1 rem*******quotes and print to ZPL port with formatting
1 rem*******************************************************

50 declare string format$(3)
60 let format$(1) = "START"
70 let format$(2) = "END"
80 let format$(3) = "DATA"
1 rem*******************************************************
1 rem main program; look for "START" keyword, if found print ^XA to ZPL port
1 rem*******************************************************

90 do
100    let begin$ = searchto$(1, format$, 2)
110    if begin$ = "START" then
120       print #2: "^XA";
1 rem*******************************************************
1 rem if "DATA" keyword is found, get two data strings
1 rem*******************************************************
130    else if begin$ = "DATA" then
1 rem*******************************************************
1 rem get data from between q
140       let extracted_data1$ = extract$(1,"""","""")
150       input #1: junk$
170       let extracted_data2$ = extract$(1,"""","""")
180       print #2:"^FO30,30^A0N,30,30^FD" &extracted_data1$& "^FS";
190       print #2:"^FO30,70^A0N,30,30^FD" &extracted_data2$& "^FS";
200    else if begin$ = "END" then
210       print #2: "^XZ"
220    end if
230 loop
```

Front Panel Control

This example shows how to intercept front panel button presses and write to the display to create a simple menu. The buttons used in this demo are set up for a Z4M/Z6M, ZM400/ZM600, or RZ400/RZ600. This could be reconfigured to work with any other printer.

Example This is an example of front panel control.

```
1 REM This example shows how to override the functionality of the feed key
1 REM and use the front panel display to show a option list
AUTONUM 1,1
REM CLOSE ALL
DECLARE STRING OPTIONS$(5)
FOR I = 1 TO 5
LET OPTIONS$(I) = "Option " & STR$(I)
NEXT I
LET ZPLPORT = 1
OPEN #ZPLPORT: NAME "ZPL"
LET FEEDKEY   = 3
LET SELECTKEY = 10
LET PLUSKEY   = 6
LET MINUSKEY  = 7
LET EXITKEY   = 9
LET TMP = REGISTEREVENT(FEEDKEY, 0, 1)
SUB NORMALLOOP
DO WHILE 1 = 1
LET EVT = HANDLEEVENT()
IF EVT = FEEDKEY THEN
LET INDEX = 1
GOSUB REGISTERKEYS
GOSUB SHOWMENU
GOTO FEEDLOOP
END IF
SLEEP 1
LOOP
SUB FEEDLOOP
DO WHILE 1 = 1
LET EVT = HANDLEEVENT()
IF EVT = FEEDKEY THEN
GOSUB RELEASEKEYS
GOSUB HIDEMENU
GOTO NORMALLOOP
ELSE IF EVT = SELECTKEY THEN
GOSUB HANDLEOPTION
ELSE IF EVT = PLUSKEY THEN
LET INDEX = INDEX + 1
IF INDEX > 5 THEN
LET INDEX = 1
END IF

GOSUB SHOWMENU
ELSE IF EVT = MINUSKEY THEN
LET INDEX = INDEX - 1
IF INDEX < 1 THEN
LET INDEX = 5
END IF
GOSUB SHOWMENU
ELSE IF EVT = EXITKEY THEN
GOSUB RELEASEKEYS
GOSUB HIDEMENU
GOTO NORMALLOOP
END IF
SLEEP 1
LOOP
REM ******** SUBROUTINE SHOWMENU ***
```

```
SUB SHOWMENU
LET LINE1$ = "FEED DISPLAY"
LET LINE2$ = OPTIONS$(INDEX)
GOSUB UPDATEDISPLAY
RETURN
REM ******** SUBROUTINE HIDEMENU ***
SUB HIDEMENU
LET LINE1$ = ""
LET LINE2$ = ""
GOSUB UPDATEDISPLAY
RETURN
SUB UPDATEDISPLAY
LET A = SETVAR("device.frontpanel.line1",LINE1$)
LET A = SETVAR("device.frontpanel.line2",LINE2$)
RETURN
SUB REGISTERKEYS
LET TMP = REGISTEREVENT(SELECTKEY, 0, 1)
LET TMP = REGISTEREVENT(PLUSKEY, 0, 1)
LET TMP = REGISTEREVENT(MINUSKEY, 0, 1)
LET TMP = REGISTEREVENT(EXITKEY, 0, 1)
RETURN
SUB RELEASEKEYS
LET TMP = UNREGISTEREVENT(SELECTKEY)
LET TMP = UNREGISTEREVENT(PLUSKEY)
LET TMP = UNREGISTEREVENT(MINUSKEY)
LET TMP = UNREGISTEREVENT(EXITKEY)
RETURN
SUB HANDLEOPTION
PRINT #ZPLPORT: "^XA^FO100,100^A0N,100,100^FD"; OPTIONS$(INDEX);"^XZ"
RETURN
```

Recall Program

This program searches for a ZPL format named "FORMAT.ZPL" that is already saved in printer memory. If the format is found, a number within the format is extracted and shown on the console. The user is then prompted to enter a new number, which is then substituted into the format.

Example This is an example of Recall.zpl

```
1 rem ************************************************************
1 rem Zebra Technologies ZBI Sample Program
1 rem
1 rem Professional programming services are available. Please contact
1 rem ZBI-Experts@zebra.com for more information.
1 rem
1 rem This is an example of recalling a ZPL format and extracting data
1 rem from it.
1 rem ************************************************************
1 rem close ports except console, open ZPL port and declare search
1 rem array
1 rem ************************************************************
10 for i = 1 to 9 step 1 ! Close all ports
20    close #i
30 next i
40 let zplport = 2
50 open #zplport: name "ZPL"
60 declare string search_zpl$(2)
70 let search_zpl$(1) = chr$(03)
80 let search_zpl$(2) = "FORMAT.ZPL"
1 rem ************************************************************
1 rem main program; look for format to recall on printer
1 rem ************************************************************
90 do
100    print #zplport: "^XA^HWE:*.ZPL^FS^XZ"
110       let present = 0
115    let find$ = ""
120    do until find$ = chr$(03)
130       let find$ = searchto$(zplport, search_zpl$)
140       if  find$ = "FORMAT.ZPL" then
150          let present = 1 ! format is present
160       end if
170    loop

1 rem ************************************************************
1 rem if format is not found, create a format and set data value to
1 rem 000
1 rem ************************************************************
180    if present = 0 then
190       print #zplport:"^XA^DFE:FORMAT.ZPL^FS";
200       print #zplport:"^FX000^FS^XZ"
210       let counter$ = "000"
1 rem ************************************************************
1 rem if format is found, extract the data from ^FX field
1 rem ************************************************************
220    else
230       print #zplport:"^XA^HFE:FORMAT.ZPL^FS^XZ"
240       let stop$ = searchto$(zplport, "^FX")
250       let counter$ = extract$(zplport, "", "^FS")
260       let stop$ = searchto$(zplport, "^XZ")
270    end if
1 rem ************************************************************
1 rem print current data value, prompt user to replace data
1 rem ************************************************************
280    print ""
290    print "Current number in format is " & counter$
300    print "Please enter new number (type EXIT to end) ";
```

```
310   input new_counter$
320   if new_counter$ = "EXIT" then
330     print "Program ending"
340     end
350   else
360     print #zplport:"^XA^DFE:FORMAT.ZPL^FS";
370     print #zplport:"^FX" & new_counter$ & "^FS^XZ"
380   end if
390 loop
```

Scale Program

This program reads data from a scale connected to the serial port by sending a "W" to the scale and waiting for a weight to be returned. When the weight is received, it is inserted into a simple label format and printed.

Example This is an example of Scale

```
1 rem ************************************************************
1 rem Zebra Technologies ZBI Sample Program
1 rem
1 rem Professional programming services are available. Please contact
1 rem ZBI-Experts@zebra.com for more information.
1 rem
1 rem This is an example of using ZBI to read scale data from the
1 rem serial port.
1 rem ************************************************************
1 rem close all ports except console, open channels to parallel and
1 rem serial ports
1 rem ************************************************************
05 for i = 1 to 9 step 1
10   close #i
20   next i
30 open # 2 : name "SER"
40 open # 1 : name "ZPL"
1 rem ************************************************************
1 rem main program; send serial port a 'W' in order to get a weight
1 rem ************************************************************
50 do
60   do
70     sleep 1   ! sleep so scale is not bombarded with incoming
1 rem data
80     print # 2 : "W" ;   ! semicolon ends sent W without a CRLF
1 rem ************************************************************
1 rem get response from scale; note that input requires a CRLF to be
1 rem entered
1 rem ************************************************************
90     input # 2 : a$
100    if a$ = "EXIT" then! back door exit - if EXIT is received, ZBI ends
110      close # 2
120      print #1: "^XZ"
130      close #1
140      end
150    end if


1 rem ************************************************************
1 rem loop until valid weight is received, then print on label
1 rem ************************************************************
160   loop while pos ( a$ , "000.00" ) = 1 or pos ( a$ , "?" ) = 1
170   print # 1 : "~SD25^XA^FS";
180   print # 1 : "^LH0,0^FS";
190   print # 1 : "^FO56,47^A0N,69,58^FDThis weighs^FS";
1 rem ************************************************************
1 rem print weight on label; & character concatenates strings
```

```
1 rem ************************************************************
200    print # 1 : "^FO56,150^A0N,69,58^FD" & A$ & " lbs^FS";
210    print # 1 : "^PQ1,0,0,N";
220    print # 1 : "^XZ"
1 rem ************************************************************
1 rem loop until weight is off scale, then repeat for next item
1 rem weighed
1 rem ************************************************************
230    do
240       print # 2 : "W" ;
250       input # 2 : A$
260    loop until pos(A$ , "000.00") = 1 or pos(A$ , "?") = 1
270 loop
```