Guide to

Converting Programs from PBasic to ZBasic

Version 1.1

Publication History

August 2008 – First draft release
September 2008 – First public release
November 2008 – Minor corrections

Disclaimer

Elba Corp. makes no warranty regarding the accuracy of or the fitness for any particular purpose of the information in this document or the techniques described herein. The reader assumes the entire responsibility for the evaluation of and use of the information presented. The Company reserves the right to change the information described herein at any time without notice and does not make any commitment to update the information contained herein. No license to use proprietary information belonging to the Company or other parties is expressed or implied.

Trademarks

Table of Contents

This page is intentionally blank.

Guide to
Converting Programs from PBasic to ZBasic

Introduction

The original BASIC programming language was designed in 1964 by John Kemeny and Thomas Kurtz at Dartmouth College. The name BASIC is commonly believed to be an acronym for **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode. The designers' intent was to create a simple, easy-to-use programming language suitable for use by non-science students. Since its conception, countless dialects and variations of the BASIC language have been developed to suit various purposes (see, for example, the Wikipedia entry http://en.wikipedia.org/wiki/List_of_BASIC_dialects).

The PBasic dialect is quite similar to the original Dartmouth BASIC. New "commands" were introduced to make it better suited to the Basic Stamp microcontrollers while a few inapplicable elements were omitted. One other difference is that Dartmouth BASIC supported operations on real numbers (i.e., floating point arithmetic) whereas PBasic only provides integer operations. PBasic is also similar to the original Dartmouth BASIC in that it lacks the concept of a callable, parameterized procedure, a much more powerful version of the GOSUB concept that appeared in later dialects.

In contrast, ZBasic is a BASIC dialect that is quite similar to the widely used Microsoft Visual Basic (VB6) language although it does not implement the entire VB6 language (e.g. variant types, Double). It also adds some useful features like Alias and Based types.

This document is intended to serve as a reference for those wanting to translate existing PBasic code for a 24-pin Basic Stamp (BS2) to the more advanced ZBasic dialect targeting a 24-pin ZX device (e.g. a ZX-24a). It will also be useful as a learning tool for those who have experience programming in PBasic and now wish to program in ZBasic. Furthermore, this document will be of benefit to those converting from MBasic and other BASIC variants that are largely PBasic compatible (e.g. PICAXE, Basic ATOM, CUBLOC, etc.). It is assumed that the reader has a comprehensive understanding of the PBasic language.

Note, particularly, that the PBasic features that are exclusive to Basic Stamp models other than the BS2 are not addressed by this document.

If you find that you have questions or issues that are not addressed by this document, please feel free to post questions on the ZBasic Forum at http://www.zbasic.net/forum or, if you prefer, send email to support@zbasic.net.

General Issues

ZBasic is inherently a procedural language, meaning that all executable code must be contained with in one or more named procedures. Every ZBasic program must have a `Main()` procedure and it is there that execution begins following power-up or a device reset. Consider the simple PBasic program below and the equivalent ZBasic program.

PBasic
```
' {$STAMP BS2}
' output a message
DEBUG "BS2 says hello, world!"
END
```

ZBasic
```
Sub Main()
  ' output a message
  Debug.Print "ZX-24a says hello, world!"
End Sub
```

There are several things worth noting about these two equivalent programs. Firstly, the PBasic compiler directive that specifies the target device is not required in ZBasic. There is a similar compiler directive in ZBasic that can be used (see `Option TargetDevice`) but it is recommended to select the target device in the IDE by using the "Device Options" dialog available from the Options menu. Secondly, there is no END statement in ZBasic; the `End Sub` construction is essentially equivalent in this context. When execution gets to the end of the `Main()` procedure, it automatically terminates (although if other tasks exist, they may continue executing). Thirdly, the indentation used on the two lines in the `Main()` procedure is not required but is commonly used to suggest the subordinate status of those lines with respect to the procedure itself. Similar indentation is commonly used with other control structures like `For…Next`, `Do…Loop`, etc. because it assists in more quickly comprehending the code structure.

ZBasic is also a modular language, meaning that the code for your application may be placed in several different files, each of which constitutes a module. The advantage of using modular programming is that you can control which aspects of the module are visible to the code in other modules of your application. Some variables and procedures in a module may be public (visible to other modules) while other variables and procedures may be private (not visible to other modules).

Other general similarities and differences between PBasic and ZBasic are listed below.

- In both PBasic and ZBasic, identifier names (labels, variables, constants, etc.) are not case sensitive and must begin with a letter followed by zero or more letters, digits or underscores. In PBasic, identifiers are limited to 32 characters while there is no limit, practically speaking, in ZBasic. As with PBasic, there is a set of reserved words in ZBasic that may not be used as identifiers in your program but the set is different from that of PBasic. See Appendix A of the ZBasic Reference Manual for the exhaustive list of reserved words.

- In ZBasic, labels must be followed by a colon as is also required in PBasic version 2.5 and later.

- In PBasic, a logical line of code may span multiple physical lines by splitting immediately after a comma. In ZBasic, the split may occur anywhere whitespace (i.e, a space or a tab) may be used but an underscore character preceded by at least one space must be present the end of each continued physical line. See Section 2.8 of the ZBasic Reference Manual for more details.

- In ZBasic, just as in PBasic, you may place multiple statements on one line by separating the statements with a colon.

One final note about procedures should be made before continuing. A ZBasic procedure is a block of code that can be invoked from various places in your program, similar in some respects to a PBasic subroutine. It can be thought of as a "black box" that implements some useful functionality. A procedure may be defined with zero or more "formal" parameters and each time it is invoked an "actual" parameter must be provided (in most cases) corresponding to each "formal" parameter. This provides a cleaner, more robust calling mechanism compared to PBasic where your only choice is to provide any necessary parameters to your subroutines using global variable values (N.B., all variables are global in PBasic).

Procedures in ZBasic come in two forms: subroutines and functions. The difference between these two forms will be explained in more detail later but the quick explanation is that a function returns a value directly to the caller while a subroutine does not. This characteristic allows a function invocation to be used in an expression in the same way that a variable or constant may be used. In the remainder of this document, we will generally refer to subroutines and functions collectively as *procedures* when the comments apply to both forms while using the words *subroutine* or *function* when the comments apply solely to that particular form. By the way, the `Sub` in the example above is a keyword indicating the definition of subroutine as opposed to a function.

I/O Ports and Pins

The 24-pin ZX devices like the ZX-24a are pin-compatible with the 24-pin Basic Stamp devices. Both have 16 I/O pins located on pins 5 through 20 although they are organized slightly differently. The 16 I/O pins on the BS2 are referred to as P0 through P15 and are accessible bit-wise, nibble-wise, byte-wise and word-wise. On the ZX-24a, the 16 I/O pins are referred to individually as pin 5 through pin 20. Pins 5 through 12 can also be manipulated as an 8-bit quantity called PortC and pins 13 through 20 can be manipulated as PortA (although it should be noted that the bit order is reversed on both ports with respect to PBasic). Nibble and word access to the two ports can be done but there are no language shortcuts provided to do so as there are in PBasic.

As in PBasic, all pins are automatically set to be inputs immediately after power-up and reset. As indicated in the table below, in PBasic pins are referred to using their logical numbers: 0-15. In ZBasic, you can refer to a pin either by its physical pin number (5-20) or by its port/bit designation which reflects both the I/O port name and the bit number in that port, e.g. `C.7`.

Basic Stamp vs ZX-24 I/O Pins

BS2 (logical pin)	ZX-24 (physical pin)
0	5, C.7
1	6, C.6
2	7, C.5
3	8, C.4
4	9, C.3
5	10, C.2
6	11, C.1
7	12, C.0
8	13, A.7
9	14, A.6
10	15, A.5
11	16, A.4
12	17, A.3
13	18, A.2
14	19, A.1
15	20, A.0

One important aspect of the table above that should be noted is that the order of bit significance is reversed between PBasic and ZBasic. This will only be significant when the I/O ports are read or written *en masse*. Individual pins are read and written in a similar manner between PBasic and ZBasic as illustrated by the simple programs below.

PBasic

```
' {$STAMP BS2}
' set pin 5 low
LOW 0
END
```

ZBasic

```
Sub Main()
  ' set pin 5 low
  PutPin 5, 0
End Sub
```

In ZBasic, the `PutPin` "command" combines the functionality of the PBasic commands `INPUT`, `LOW`, `HIGH` and `TOGGLE` and includes additional functionality such as configuring the pin's pull-up resistor (when configuring as an input) and pulsing the output. The first parameter indicates the pin number while the second parameter is a "mode" value that indicates the desired action as shown in the table below. Note the pre-defined constants that correspond to each of the mode values enumerated in the table.

Mode Values for `PutPin`

Value	Built-in Constant	Effect
0	`zxOutputLow`	The pin is an output at logic zero.
1	`zxOutputHigh`	The pin is an output at logic one.
2	`zxInputTriState`	The pin is an input with the pull-up resistor disabled.
3	`zxInputPullUp`	The pin is an input with the pull-up resistor enabled.
4	`zxOutputToggle`	Change the logic level of the output.
5	`zxOutputPulse`	Output a pulse.

In the discussion above, PutPin was described as a “command” but in reality it is a built-in subroutine, part of the ZBasic System Library. In ZBasic, any subroutine may be invoked merely by giving its name and following that with a comma separated list of parameters, if any. This style is similar to the command style of PBasic but it is anachronistic. The more modern style is to use the keyword Call before the subroutine name and then to put parentheses around the parameter list (even if there are no parameters). The ZBasic sample code for setting a pin low is shown below rewritten in this style and also using the built-in constant for the mode parameter.

ZBasic

```
Sub Main()
  ' set pin 5 low
  Call PutPin(5, zxOutputLow)
End Sub
```

It is strongly suggested that you use defined constants instead of numeric constants for elements like the pin number in the PutPin() invocation above. Assuming that the constant name is suggestive of how the pin is being used, doing so makes the code easier to understand. Moreover, if the same pin is used in many places, using a defined constant makes maintenance and modification of the code much simpler and less error prone because you need only change the pin number in one place – in the constant definition.

As mentioned earlier, all I/O pins are automatically configured to be inputs (with the pull-up resistor disabled) immediately after power up and after a reset. When you invoke the PutPin() subroutine, the individual pin involved will be reconfigured as necessary according to the value of the second parameter, the mode value. I/O pins may also be reconfigured *en masse* in both PBasic and ZBasic by writing values to special “registers”. In PBasic, the I/O pins are configured eight at a time using the special memory locations OUTL, OUTH, DIRL and DIRH. In ZBasic, the idea is the same but the syntax is slightly different as shown in the table below.

I/O Pin Special Registers

PBasic	ZBasic
INL	Register.PinC
INH	Register.PinA
OUTL	Register.PortC
OUTH	Register.PortA
DIRL	Register.DDRC
DIRH	Register.DDRA

For the most part, I/O pin configuration details are the same between PBasic and ZBasic. It should be noted, however, that the bit order is reversed between the two systems. In PBasic, the least significant bit of OUTL corresponds to pin 5 while in ZBasic, the most significant bit of Register.PortC corresponds to pin 5.

As in PBasic, when a bit in the data direction register (e.g. DIRL or Register.DDRC) is a zero, the corresponding pin will be an input and when a data direction register bit is one, the corresponding pin will be an output. Also like PBasic, when a pin is an output, the corresponding bit in the output register (e.g. OUTL or Register.PortC) controls the output state, zero or one. However, unlike PBasic, in ZBasic when a pin is configured as an input the corresponding pin in the output register controls whether that pin’s pull-up resistor is enabled or not. If the bit is a one the pull-up resistor is enabled, if it is zero the pull-up resistor is disabled. The latter condition is referred to variously as the “high impedance input” mode, the “high-Z input” mode, or the “tri-state input” mode.

Defining and Using Constants

There are several benefits of defining and using constants as alluded to in the preceding section. The process of defining and using a constant is fundamentally the same in PBasic and ZBasic but the details differ slightly. Consider this PBasic constant definition.

PBasic

```
' define the pin used to drive the LED
ledPin CON 6
```

The definition in ZBasic, shown below, contains the same elements but in a different order and using a different keyword. The ZBasic constant definition has one additional element not present in the PBasic definition – the type of the constant.

ZBasic

```
' define the pin used to drive the LED
Const ledPin as Byte = 6
```

Unlike PBasic, ZBasic is a strongly typed language. This means that the ZBasic compiler does not silently convert data of one type to another type and does not allow a data item of one type to be assigned to a data element having a different type. Implicit type conversions are commonly considered undesirable because they can lead to unexpected problems when the programmer is not even aware that a conversion occurred or what the technical details of the conversion might be. This is much less of a problem in PBasic because it has only a few data types; unsigned integral types for the most part. Conversions between different widths of unsigned integral types are fairly straightforward, using simple rules and leading to few, if any, surprises. In contrast, ZBasic has a much richer set of data types available including both signed and unsigned integral types, a Boolean type, a real number type, a true string type, user-defined enumerated types, user-defined composite types (structures) and other types. Given this wide range of types with differing characteristics, it is preferable to require all program elements to have a declared type and to require explicit type conversions rather than allowing the compiler to perform them implicitly. The ZBasic System Library provides a series of functions useful for converting one data type to another. For example, the `CInt()`function converts the parameter provided to an `Integer` type.

As in PBasic, the value associated with a constant may be specified using a simple value or a more complex expression involving specific values, other constants and various operators. In ZBasic, you may also use many ZBasic System Library functions in constant value expressions provided that the parameters to the function, if any, are also constant.

In ZBasic a constant may be defined at the module level as either public or private (see the examples below). If it is public, code in other modules will “see” the constant and may use it. If it is private, only the code in the module where it is defined may use the constant. For constants defined within a procedure, the constant is visible only to code within that procedure.

ZBasic

```
Private Const ledPin as Byte = 6
Public Const rowCount as Integer = 7
```

One last difference between PBasic and ZBasic in defining constants is that the definition of any constant in ZBasic must precede its first use in the module where it is defined.

Using Numeric Constants

In PBasic, numeric constants may be specified in several different ways: decimal, hexadecimal, binary and character. The same capability exists in ZBasic using somewhat different methods as shown in the table below.

Specifying Numeric Contants

PBasic	ZBasic	Description
`99`	`99`	decimal
`$FE`	`&HFE`	hexadecimal
`%1010`	`&B1010`	binary
`"A"`	`Chr("A")`	character
	`"Hello, world!"`	string
	`3.14159`	real number, decimal
	`6.02E23`	real number, scientific notation

Defining and Using Variables

When you define a variable in PBasic, the choice of the variable's type is limited to the types shown in the leftmost column of the table below. ZBasic supports those types but also supports other useful types as shown in the second column of the table.

Available Data Types

PBasic	ZBasic	Description
BIT	Bit	1 bit, unsigned
NIBBLE	Nibble	4 bits, unsigned
BYTE	Byte	8 bits, unsigned
WORD	UnsignedInteger	16 bits, unsigned
	Integer	16 bits, signed
	UnsignedLong	32 bits, unsigned
	Long	32 bits, signed
	Single	32 bits, floating point
	String	0-255 characters

The syntax for defining a variable is slightly different between PBasic and ZBasic as shown in the examples below.

PBasic
```
' define a variable for the counter
cnt VAR BYTE
```

ZBasic
```
' define a variable for the counter
Dim cnt as Byte
```

In both PBasic and ZBasic, a variable may be defined as an array, that is, containing multiple elements of the same type.

PBasic
```
' define an array for the data values
dataVal VAR BYTE(4)
```

ZBasic
```
' define an array for the data values
Dim dataVal(4) as Byte
```

The PBasic example above defines an array of four byte-size elements that are accessed using the indices 0-3. The meaning of the ZBasic example is similar but slightly different. ZBasic is much more flexible in that you can specify both the lower bound and the upper bound of the index range. By default, unless you specify a lower bound explicitly the lower bound is zero. Consequently, the ZBasic example above defines an array of five byte-size elements that are accessed using the indices 0-4. To get the same effect as the PBasic example, you need to specify the upper bound that you want.

ZBasic
```
' define an array for the data values with four elements
Dim dataVal(3) as Byte
```

Alternately, you may specify both the lower bound and the upper bound:

ZBasic
```
' define an array for the data values with four elements
Dim dataVal(0 to 3) as Byte
```

In ZBasic, the lower and upper bound values may be specified using constant expressions and the values may be any valid integral value, even negative values. The only requirement is that the upper bound must be greater

than or equal to the lower bound. Although the default lower bound is zero, it is quite common for ZBasic programs to written with an explicit lower bound of 1.

ZBasic also supports multi-dimensional arrays; the additional lower/upper bound specifications are added inside the parentheses with a comma separating each pair.

ZBasic
```
' define a two-dimension array for the data values
Dim dataVal(1 to 4, 1 to 5) as Byte
```

One subtle difference when accessing array values is that in PBasic, you may refer to the first element of the array (i.e. with index 0) by specifying only the array name. In ZBasic, array indices are always required; the compiler will emit an error message indicating that an array index expression is required if you omit it.

In PBasic, you can define an additional variable that occupies the same space as another variable; this is called an alias. You can do the same in ZBasic as shown in the examples below.

PBasic
```
' define two variables that use the same space
cnt VAR Integer
idx VAR cnt.LOBYTE
```

ZBasic
```
' define two variables that use the same space
Dim cnt as Integer
Dim idx as Byte Alias cnt
```

In the alias examples above, the second variable is aligned with the low address of the first variable. In PBasic, you can define an alias that is aligned anywhere within the host variable using similar syntax as shown in the example below. In ZBasic, an alias is always aligned with the low byte of the host variable. You can use a based variable accomplish arbitrary alignment in ZBasic as shown in the example below. The expression following the `Based` keyword specifies the address where you want the variable located. The construction `cnt.DataAddress` evaluates to the address of the variable `cnt`. For more information about based variables see section 3.22 in the ZBasic Reference Manual.

PBasic
```
' define two variables that use the same space
cnt VAR Integer
idx VAR cnt.HIBYTE
```

ZBasic
```
' define two variables that use the same space
Dim cnt as Integer
Dim idx as Byte Based cnt.DataAddress + 1
```

In ZBasic a variable may be defined at the module level as either public or private (see the examples below). If it is public, code in other modules will “see” the variable and may access it. If it is private, only the code in the module where it is defined may use the variable. For variables defined within a procedure, the variable is visible only to code within that procedure.

ZBasic
```
Private cnt as Integer
Public idx as byte
```

One last difference between PBasic and ZBasic in defining variables is that the definition of any variable in ZBasic must precede its first use in the module where it is defined.

Expressions, Operators and Order of Operations

As noted earlier, the PBasic compiler allows mixing of different operand types in expressions and performs automatic type conversions. In contrast, since ZBasic is a strongly typed language the ZBasic compiler generally requires agreement between the types of operands supplied and the type expected by the operator or function. In most cases, it is a simple matter to insert the appropriate conversion function to ensure type agreement.

Both PBasic and ZBasic support the same set of basic arithmetic operators that can be used in expressions. Beyond the basic operators, each supplies a similar set of operators with a few differences here and there. The table below indicates the available unary operators, i.e. those that take one operand.

Unary Expression Operators

PBasic	ZBasic	Description
-	-	Negation
~	Not	Logical or bit-wise complement
NCD		priority encoder
DCD		decoder
ABS	Abs	absolute value
SQR	Sqr	square root
HYP		hypotenuse
SIN	Sin	trigonometric sine
COS	Cos	trigonometric cosine
	Tan	trigonometric tangent
	ASin	trigonometric inverse sine
	ACos	trigonometric inverse cosine
ATN	ATan	trigonometric inverse tangent

For ZBasic, the "operators" in the table above (other than `Not`) which are in the form of an identifier (e.g. `Abs`) are actually functions that are described in detail in the ZBasic System Library Reference Manual. There are some differences in the details of these operators between PBasic and ZBasic. For example, in ZBasic, the `Sqr()` function only operates on real numbers. You can still use it with integral values but you must first convert the operand to type `Single` and then convert the result back to the desired integral type. For example:

ZBasic

```
' take the square root of a byte value
Dim val as Byte
val = CByte(Sqr(CSng(val)))
```

Also, the trigonometric functions in ZBasic utilize radian angle measure while in PBasic they utilize "binary radian" angle measure. Moreover, the ZBasic trigonometric functions accept only type `Single` parameters and return type `Single` results. See the corresponding entries in the ZBasic System Library Reference Manual for complete details.

The table below indicates the available binary expression operators (i.e. those that require two operands). As noted earlier, the entries that are in the form of an identifier (other than `Mod`, `And`, `Or` and `Xor`) are actually functions that are described in full detail in the ZBasic System Library Reference Manual.

Binary Expression Operators

PBasic	ZBasic	Description
+	+	addition
-	-	subtraction
*	*	multiplication
**		multiply high
*/		multiply middle
/	\	division, integral
	/	division, real number

//	Mod	modulus
&	And	logical or bit-wise AND
|	Or	logical or bit-wise OR
^	Xor	logical or bit-wise XOR
<<	Shl	left shift
>>	Shr	right shift
MIN	Max	the largest of two values
MAX	Min	the smallest of two values
DIG		return a specified digit of a value
REV		reverse the order of a series of bits
DCD		
	&	string concatenation

In PBasic, expression evaluation is performed strictly in left to right order unless other ordering is forced by the use of parentheses. In ZBasic, you may use parentheses to force a particular evaluation order but in their absence the order of operations is governed by an operator precedence system as described in Section 2.4.1 of the ZBasic Language Reference Manual.

Defining and Using Subroutines

A subroutine is a collection of statements that can be executed by invoking it from various places in your program, optionally passing parameter values that may be different on each invocation. The advantage of creating subroutines is that they can thought of as “black boxes” that perform a specific function. A subroutine may use its own local variables and constants and may call other subroutines as necessary.

A simple example of a subroutine is given below and other examples can be found in the PBasic Conversion Helper Code section of this document. A complete discussion of subroutines is presented in Section 2.3.2 of the ZBasic Language Reference Manual.

ZBasic

```
'
'' displayValue
'
' Output a message with a name/value pair.
'
Sub displayValue(ByVal name as String, ByVal val as UnsignedInteger)
  Debug.Print "The value of "; name; " is "; val
End Sub

Sub Main()
  Call displayValue("weight", 100)
  Call displayValue("length", 12)
End Sub
```

Defining and Using Functions

A function in ZBasic is essentially the same as a subroutine except that a function returns a value when it is invoked. This characteristic allows a function invocation to be used in an expression in place of a variable or constant value.

A simple example of a function is given below and other examples can be found in the PBasic Conversion Helper Code section of this document. A complete discussion of functions is presented in Section 2.3.2 of the ZBasic Language Reference Manual.

ZBasic

```
'
'' cmToInches
'
' Convert a length in centimeters to the equivalent in inches.
'
Function cmToInches(ByVal cm as Single) as Single
  cmToInches = cm / 2.54
End Function

Sub Main()
  Dim inches as Single
  Dim cm as Single
  cm = 80.5
  inches = cmToInches(cm)
End Sub
```

PBasic Statements, Commands and Directives

This section deals with issues related to converting the various PBasic statements, commands and directives to equivalent ZBasic code. Inasmuch as several of the control type statements do not adhere to structured programming precepts, alternatives are suggested to produce a more structured program.

BRANCH

This control statement, essentially a computed goto, clearly does not conform to structured programming guidelines. Fortunately, it is fairly straightforward to implement the control logic of a BRANCH statement using the `Select Case` control structure of ZBasic. Consider the simple BRANCH example below and the suggested replacement.

PBasic

```
val VAR BYTE
  BRANCH val, lab1, lab2, lab2, lab3
lab1:
  val = 25
  GOTO next
lab2:
  val = 18
  GOTO next
lab3:
  val = 100
next:
```

ZBasic

```
Sub Main()
  Dim val as Byte

  Select Case val
  Case 0
    val = 25
  Case 1, 2
    val = 18
  Case 3
    val = 100
  End Select
End Sub
```

To handle special cases where control falls through from one label to another in the PBasic code, you'll either have factor out the common code into a separate procedure, duplicate the code in the multiple cases or add a label in the destination case and jump to it from the end of fall-through case.

It should be noted that the logic of the `Select Case` control structure can be equally well implemented using an `If…Then…Else` control structure. This is illustrated in the alternate implementation below.

ZBasic

```
Sub Main()
  Dim val as Byte

  If (val = 0) Then
    val = 25
  ElseIf ((val = 1) Or (val = 2)) Then
    val = 18
  Else (val = 3) Then
    val = 100
  End If
End Sub
```

BUTTON

There is no ZBasic System Library routine that directly implements the functionality of the `BUTTON` command. However, part of the functionality can be fairly easily implemented. See the `Button()` function in the PBasic Conversion Helper Code set.

COUNT

The functionality of this command can be implemented using the ZBasic System Library function `CountTransitions()`. However, there are some differences in the functionality of the `COUNT` command and the `CountTransitions()` function that must be noted.

.

COUNT	**CountTransitions()**
Counts full cycles.	Counts logic transitions (two per cycle).
Automatically makes the specified pin an input.	The specified pin must be made an input before calling.
The counting period has units of milliseconds.	The counting period has units of I/O Timer tick period (default: 2.441uS) or seconds depending on the expression type.

PBasic

```
  myPin CON 0
  countPeriod CON 100
  cnt VAR WORD
  COUNT myPin, countPeriod, cnt
```

ZBasic

```
Sub Main()
  Const countPeriod as Single = 100.0E-3
  Const myPin as Byte = 5
  Dim cnt as Long
  Call PutPin(myPin, zxInputTriState)
  cnt = CountTransitions(myPin, countPeriod) \ 2
End Sub
```

DATA

There is no direct equivalent of the `DATA` statement in ZBasic. However, in most cases it should be possible to implement the same functionality using an initialized Program Memory data item in ZBasic. A simple example involving Byte data is shown below.

PBasic

```
myData DATA 100, 200, 52, 45
```

ZBasic

```
Dim myData as ByteVectorData({ 100, 200, 52, 45 })
```

Note that the ZBasic compiler will automatically assign a Program Memory address to the `myData` data item; there is no way to control the address directly as there is in PBasic. Also, you cannot mix 8-bit and 16-bit data items in ZBasic as you can in PBasic. See the discussion of Program Memory Data Items in section 2.10 of the ZBasic Reference Manual for more information.

DEBUG

The closest equivalent to the PBasic `DEBUG` command is the `Debug.Print` statement in ZBasic. However, `Debug.Print` does not have the multitude of data conversion/formatting options available with `DEBUG`. Instead, `Debug.Print` accepts string arguments only. You'll have to use other ZBasic functions to convert the data items into strings.

DEBUGIN

The closest equivalent to the PBasic `DEBUGIN` command is the `Console.Read` or `Console.ReadLine` statement in ZBasic. However, neither of these support the data conversion options available with `DEBUGIN`. You'll have to use other ZBasic functions to convert the received strings into data values.

DO…LOOP

The five `Do…Loop` variants available in PBasic correspond exactly to the five variants available in ZBasic. The only difference in practice is the statement used to prematurely terminate the loop; in PBasic the `EXIT` statement is used while in ZBasic it is `Exit Do`.

DTMFOUT

The application note AN-201 "Generating DTMF Using FreqOut" (available at http://www.zbasic.net) contains a subroutine `DTMF()` that can be used as a building block to implement the functionality of `DTMFOUT`.

END

The PBasic END statement has two effects. Firstly, execution of the main program ceases and secondly, the device enters a low-power mode. In ZBasic, the first effect is implemented automatically at the end of the `Main()` subroutine. If the second effect is needed, it can be obtained using the `Sleep()` subroutine in the ZBasic System Library.

FOR…NEXT

The `For…Next` control constructs in PBasic and ZBasic are similar. The differences are summarized in the table below.

.

PBasic	ZBasic
A decrementing For loop is created by specifying a start value larger than the end value.	A decrementing For loop is created by specifying a negative step value. Generally, the control variable should be a signed type (i.e. `Integer`) in this case.
Premature loop termination is effected by the `EXIT` statement.	Premature loop termination is effected by the `Exit For` statement.
The loop control variable name may not be added after the `NEXT` keyword.	The loop control variable name optionally may be added after the `Next` keyword.

FREQOUT

The functionality of this command can be implemented using the ZBasic `FreqOut()` subroutine. However, there are some differences in the functionality of the `FREQOUT` command and the `FreqOut()` subroutine that must be noted, shown in the table below.

FREQOUT	FreqOut()
Temporarily makes the specified pin an output.	The specified pin is configured as an output and left that way.
The duration parameter has units of milliseconds.	The duration paremeter has units of seconds or milliseconds depending on the data type of the expression.
The second frequency is optional.	Two frequencies must be specified but they may be identical.

The example code below illustrates how to write equivalent code.

PBasic
```
  myPin CON 0
  duration CON 500
  freq1 CON 440
  freq2 CON 880
  FREQOUT myPin, duration, freq1, freq2
```

ZBasic
```
Sub Main()
  Const myPin as Byte = 5
  Const freq1 as Integer = 440
  Const freq2 as Integer = 880
  Const duration as Single = 500.0E-3
  Call FreqOut(myPin, freq1, freq2, duration)
End Sub
```

GOSUB

Although not a direct equivalent, the `Call` statement in ZBasic is similar in functionality to the `GOSUB` statement.

PBasic
```
  GOSUB mySub
  <other code>

mySub:
  <other code>
  RETURN
```

ZBasic
```
Sub Main()
  Call mySub()
  <other code>
End Sub

Sub mySub()
  <other code>
End Sub
```

GOTO

The `GoTo` statement in ZBasic is identical to the `GOTO` statement in PBasic in functionality. It is recommended, however, that the `GoTo` statement be used as little as possible, preferably not at all. In most cases, equivalent logic can be implemented in ZBasic without using a `GoTo` and in most cases the resulting code is easier to comprehend.

HIGH

In ZBasic, an I/O pin is configured using the PutPin() subroutine as illustrated in the example below.

PBasic
```
  myPin CON 0
  HIGH myPin
```

ZBasic
```
Sub Main()
  Const myPin as Byte = 5
  Call PutPin(myPin, zxOutputHigh)
End Sub
```

IF…THEN

The syntax of the ZBasic `If…Then` control construct is nearly identical to the newer style PBasic `IF…THEN` supported in V2.5 and later. The only difference is that the `ENDIF` keyword is used in PBasic to terminate the construct while the keyword sequence `End If` is used in ZBasic.

PBasic
```
myPin CON 0
IF myVar > 10 THEN
  HIGH myPin
ELSE
  LOW myPin
ENDIF
```

ZBasic
```
Sub Main()
  Const myPin as Byte = 5
  If (myVar > 10) Then
    Call PutPin(myPin, zxOutputHigh)
  Else
    Call PutPin(myPin, zxOutputHigh)
  End If
End Sub
```

The single line form of `IF…THEN` is not supported by ZBasic. Such constructs in existing code will need to be changed to the multi-line form.

INPUT

The functionality of this command can be realized in ZBasic using the `PutPin()` subroutine as illustrated in the example below. Note that you also have the option in ZBasic of enabling the internal pull-up resistor for the pin.

PBasic
```
  myPin CON 0
  INPUT myPin
```

ZBasic
```
Sub Main()
  Const myPin as Byte = 5
  Call PutPin(myPin zxInputTriState)
End Sub
```

LOOKDOWN

The LOOKDOWN statement has no direct equivalent in ZBasic. However, the same logic can be implemented using a Select Case or If…Then control construct as illustrated in the example below.

PBasic
```
val VAR BYTE
idx VAR BYTE
val = 17
idx = 255
LOOKDOWN val, >[26, 177, 13], idx
DEBUG "item ", DEC idx
```

ZBasic
```
Sub Main()
  Dim val as Byte
  Dim idx as Byte
  val = 17
  idx = 255
  If (val > 26) Then
    idx = 0
  ElseIf (val > 177) Then
    idx = 1
  ElseIf (val > 13) Then
    idx = 2
  End If
  Debug.Print "item "; idx
End Sub
```

If the values are constant, an alternate implementation strategy is to use an initialized Program Memory data item and loop through the values.

ZBasic
```
Sub Main()
  Dim val as Byte
  Dim idx as Byte
  val = 17
  idx = 255
  Dim valList as ByteVectorData({ 26, 177, 13 })
  Dim i as Integer
  For i = 1 to Ubound(valList)
    If (val > valList(i)) Then
      idx = LoByte(i) - 1
      Exit For
     End If
  Next i
  Debug.Print "item "; idx
End Sub
```

LOOKUP

The LOOKUP statement has no direct equivalent in ZBasic. However, the same logic can be implemented using the Select Case or If…Then control construct as illustrated in the example below.

PBasic
```
val VAR BYTE
idx VAR BYTE
val = 255
idx = 1
LOOKUP idx, [26, 177, 13], val
```

```
DEBUG "item ", DEC idx, " is ", DEC val
```

ZBasic
```
Sub Main()
  Dim val as Byte
  Dim idx as Byte
  val = 255
  idx = 1
  Select Case idx
  Case 0
    val = 26
  Case 1
    val = 177
  Case 2
    val = 13
  End Select
  Debug.Print "item "; idx; " is "; val
End Sub
```

If the values are constant, an alternate implementation strategy is to use an initialized Program Memory data item and access the corresponding value by index.

ZBasic
```
Sub Main()
  Dim val as Byte
  Dim idx as Byte
  Val = 255
  idx = 1
  Dim valList as ByteVectorData({ 26, 177, 13 })
  If (idx < CByte(Ubound(valList))) Then
    val = valList(idx + 1)
  End If
  Debug.Print "item "; idx; " is "; val
End Sub
```

LOW

The functionality of this command can be realized in ZBasic using the `PutPin()` subroutine as illustrated in the example below.

PBasic
```
  myPin CON 0
  LOW myPin
```

ZBasic
```
Sub Main()
  Const myPin as Byte = 5
  Call PutPin(myPin, zxOutputLow)
End Sub
```

NAP

There is no direct equivalent of this command in ZBasic. The functionality can be approximated using the `Nap()` subroutine in the PBasic Conversion Helper Code set.

PBasic
```
  NAP 3
```

ZBasic
```
Sub Main()
```

```
  Call Nap(3)
End Sub
```

ON…GOSUB

This control structure, introduced in PBasic v2.5, is nearly identical to the BRANCH statement, the difference being that the code at the destination labels is expected to RETURN and execution will continue with the line immediately following the ON…GOSUB. See the discussion of the BRANCH statement for conversion suggestions.

ON…GOTO

This control structure, introduced in PBasic v2.5, is identical in functionality to the BRANCH statement. See the discussion there for conversion suggestions.

OUTPUT

The functionality of this command can be realized in ZBasic by manipulating the data direction register directly as illustrated in the example below. Generally, however, it is simpler to just use PutPin() to also set the output state at the same time.

PBasic
```
  myPin CON 0
  OUTPUT myPin
```

ZBasic
```
Sub Main()
  Const myPin as Byte = 5
  Call SetBits(Register.DDR(myPin), PortMask(myPin), &Hff)
End Sub
```

PAUSE

The corresponding functionality for PAUSE can be implemented using the ZBasic System Library subroutine Sleep() but the parameter value must be converted to seconds as shown below.

PBasic
```
  pauseTimeMS CON 55
  PAUSE pauseTimeMS
```

ZBasic
```
Sub Main()
  Const pauseTimeMS as UnsignedInteger = 55
  Call Sleep(CSng(pauseTimeMS) / 1000.0)
End Sub
```

Note that in the example above, the compiler will convert the parameter to Sleep() to the approximately equivalent number of RTC ticks which occur every 1.95mS. If the pause time is not constant, the compiler will generate code to perform the conversion to RTC ticks at run-time.

If more precise pauses are required, consider using the ZBasic System Library subroutine PulseOut() as illustrated below.

ZBasic
```
Sub Main()
  Const pauseTimeMS as UnsignedInteger = 55
  Call PulseOut(0, CSng(pauseTimeMS) / 1000.0, 0)
End Sub
```

PULSIN

The functionality for PULSIN can be approximated using the ZBasic System Library subroutine PulseIn() as shown below. Note that the maximum pulse width for the ZBasic equivalent as written is about 35.5mS. However, the time base can be adjusted to yield longer maximum pulse widths. See the description of PulseIn() for more details.

PBasic

```
  myPin CON 0
  pulseWidth VAR WORD
  PULSIN myPin, 1, pulseWidth
```

ZBasic

```
Sub Main()
  Const pwUnits as Single = 1.085E-6
  Const myPin as Byte = 5
  Dim pulseWidth as UnsignedInteger
  Dim pw as Single
  Call PulseIn(myPin, 1, pw)
  pulseWidth = CUInt(pw / pwUnits / 2.0)
End Sub
```

An alternate implementation uses the function form of PulseIn() which returns an Integer value.

ZBasic

```
Sub Main()
  Const pwUnits as Single = 1.085E-6
  Const myPin as Byte = 5
  Dim pulseWidth as UnsignedInteger
  pulseWidth = CUInt(CSng(PulseIn(myPin, 1)) / pwUnits / 2.0)
End Sub
```

PULSOUT

The functionality for PULSOUT can be approximated using the ZBasic System Library subroutine PulseOut() but the parameter value must be converted to seconds as shown below. Note, also, that a third parameter must be provided – the desired pulse active state. In some cases, using PutPin() with a second parameter of zxOutputPulse may be appropriate but the pulse width in that case will be approximately 500nS.

PBasic

```
  myPin CON 0
  pulsePeriod CON 5
  LOW myPin
  PULSOUT myPin, pulsePeriod
```

ZBasic

```
Sub Main()
  Const myPin as Byte = 5
  Const pulsePeriod as UnsignedInteger = 5
  Call PutPin(myPin, zxOutputLow)
  Call PulseOut(myPin, CSng(pulsePeriod) * 2.0 / 1.0E6, 1)
End Sub
```

PWM

The functionality of `PWM` can be approximated using the ZBasic System Library subroutine `PutDAC()` as shown below. Note that a PWM cycle in ZBasic lasts about 200uS compared to approximately 1mS in PBasic. This limits the effective range of the 'cycles' parameter to 1/5 of that in PBasic. Note also, that depending on the requirements of the application it may be a better solution to use the true PWM routines in the ZBasic System Library.

PBasic

```
  myPin CON 0
  duty CON 128
  cycles CON 10
  PWM myPin, duty, cycles
```

ZBasic

```
Sub Main()
  Const myPin as Byte = 5
  Const duty as Byte = 128
  Const cycles as Byte = 10
  Dim dacAcc as Byte
  Call PutDAC(myPin, duty, dacAcc, cycles * 5)
End Sub
```

RANDOM

There is no direct replacement for the PBasic `RANDOM` statement in ZBasic. The effect can be approximated as shown below.

PBasic

```
  rndVal VAR WORD
  RANDOM rndVal
```

ZBasic

```
Sub Main()
  Dim rndVal as UnsignedInteger
  rndVal = CUInt(Rnd() * 65535.0)
End Sub
```

RCTIME

The functionality for `RCTIME` can be approximated using the ZBasic System Library subroutine `RCTime()` as shown below. The application note AN-202 "Using RCTime to Measure Charging Time" (available at http://www.zbasic.net) gives more information about using the `RCTime()` function. One difference is that the units for the returned value in PBasic are 2uS while in ZBasic the units are 1.085uS.

PBasic

```
  myPin CON 0
  chargeTime VAR WORD
  HIGH myPin
  PAUSE 1
  RCTIME myPin, 1, chargeTime
```

ZBasic

```
Sub Main()
  Const rcTimeUnits as Single = 1.085E-6
  Const myPin as Byte = 5
  Dim chargeTime as UnsignedInteger
  Dim ct as Single
```

```
  Call PutPin(myPin, 1)
  Call Sleep(1)
  Call RCTime(myPin, 1, ct)
  chargeTime = CUInt(ct / rcTimeUnits / 2.0)
End Sub
```

An alternate implementation uses the function form of `RCTime()` which returns an `Integer` value.

ZBasic
```
Sub Main()
  Const rcTimeUnits as Single = 1.085E-6
  Const myPin as Byte = 5
  Dim chargeTime as UnsignedInteger
  Call PutPin(myPin, 1)
  Call Sleep(1)
  chargeTime = CUInt(CSng(RCTime(myPin, 1)) / rcTimeUnits / 2.0)
End Sub
```

READ

The functionality of the PBasic `READ` command can be implemented using the ZBasic System Library subroutine `GetProgMem()`. It should be noted, however, that reading from or writing to arbitrary locations in Program Memory is not advised. In most cases, using initialized Program Memory data items and the normal accessing mechanism for them will serve your purposes. See the discussion of Program Memory Data Items in section 2.10 of the ZBasic Reference Manual for more information.

PBasic
```
  val VAR BYTE
  READ 0, val   ' read 1 byte from Program Memory location 0
```

ZBasic
```
Sub Main()
  Dim val as Byte
  Call GetProgMem(0, val, 1)  ' read 1 byte from Program Memory location 0
End Sub
```

RETURN

The PBasic RETURN command has no direct equivalent in ZBasic; it is not needed. A return from a procedure is implied by the end of the procedure definition, and `Exit Sub` statement or an `Exit Function` statement.

REVERSE

The functionality of this command can be realized in ZBasic by manipulating the data direction register directly as illustrated in the example below. Generally, however, it is simpler to just use `PutPin()` to set the pin configuration.

PBasic
```
  myPin CON 0
  REVERSE myPin
```

ZBasic
```
Sub Main()
  Const myPin as Byte = 5
  Call SetBits(Register.DDR(myPin), PortMask(myPin), Not Register.DDR(myPin))
End Sub
```

SELECT…CASE

The syntax of the ZBasic `Select Case` control construct is nearly identical to that of PBasic (supported in V2.5 and later). The differences are summarized in the table below.

PBasic	ZBasic
The construct begins with `Select <expr>`.	The construct begins with `Select Case <expr>`.
A conditional case value is expressed using just a conditional operator and an expression value.	A conditional case value is expressed using the keyword `Is`, a conditional operator and an expression value.
The construct ends with `ENDSELECT`.	The construct ends with `End Select`.

PBasic

```
myPin Con 0
myVar VAR WORD
SELECT myVar
CASE 0
  HIGH myPin
CASE 1 TO 9
  myVar = 100
CASE < 100
  LOW myPin
ENDSELECT
```

ZBasic

```
Dim myVar as Byte
Sub Main()
  Const myPin as Byte = 5
  Select Case myVar
  Case 0
    Call PutPin(myPin, zxOutputHigh)
  Case 1 to 9
    myVar = 100
  Case Is < 100
    Call PutPin(myPin, zxOutputLow)
  End Select
End Sub
```

SERIN

There is no direct equivalent for the PBasic `SERIN` command in ZBasic. In ZBasic, all serial communication is interrupt driven using queues to hold received data so that no input is lost (as long as the queue is not allowed to fill up). In contrast, data that arrives on the input pin in PBasic while the program is busy with other things is completely missed.

See the description of the ZBasic System Library routines `OpenQueue()`, `DefineCom()`, and `OpenCom()` for more information.

SEROUT

There is no direct equivalent for the PBasic `SEROUT` command in ZBasic. In ZBasic, all serial communication is interrupt driven using queues to hold received data to be transmitted so that the program doesn't have to wait during transmission.

See the description of the ZBasic System Library routines `OpenQueue()`, `DefineCom()`, and `OpenCom()` for more information.

SHIFTIN

The functionality of the PBasic `SHIFTIN` command can be realized using the ZBasic System Library function `ShiftInEx()` as shown below.

PBasic
```
  dPin CON 0
  cPin CON 1
  val VAR WORD
  SHIFTIN dPin, cPin, MSBPRE, [val\8]
```

ZBasic
```
Sub Main()
  Const dPin as Byte = 5
  Const cPin as Byte = 6
  Dim val as UnsignedInteger
  val = ShiftInEx(dPin, cPin, 8, &H02)
End Sub
```

Note that the shift clock frequency for the BS2 is approximately 16.7KHz while the default shift clock frequency for the ZX-24a is about 400KHz. If that clock speed is too fast, a fifth parameter may be added to the call to specify the desired bit time. See the discussion of `ShiftInEx()` in the ZBasic System Library Reference Manual.

SHIFTOUT

The functionality of the PBasic `SHIFTOUT` command can be realized using the ZBasic System Library subroutine `ShiftOutEx()` as shown below.

PBasic
```
  dPin CON 0
  cPin CON 1
  mode CON 0
  val VAR WORD
  val = $55
  SHIFTOUT dPin, cPin, MSBPRE, [val\8]
```

ZBasic
```
Sub Main()
  Const dPin as Byte = 5
  Const cPin as Byte = 6
  Dim val as UnsignedInteger
  val = &H55
  Call ShiftOutEx(dPin, cPin, 8, val, &H02)
End Sub
```

Note that the shift clock frequency for the BS2 is approximately 16.7KHz while the default shift clock frequency for the ZX-24a is about 400KHz. If that clock speed is too fast, a sixth parameter may be added to the call to specify the desired bit time. See the discussion of `ShiftInEx()` in the ZBasic System Library Reference Manual.

SLEEP

There is no direct equivalent for the PBasic `SLEEP` command in ZBasic. The example code below illustrates how a similar effect can be achieved although the low power consumption aspect of the `SLEEP` command is not replicated.

PBasic
```
  sleepSec CON 5
  SLEEP sleepSec
```

ZBasic
```
Sub Main()
  Const sleepSec as UnsignedInteger
  Call Sleep(CSng(sleepSec))
End Sub
```

STOP

There is no direct equivalent for the PBasic `STOP` command in ZBasic. The example code below illustrates how a similar effect can be achieved.

PBasic
```
STOP
```

ZBasic
```
Sub Main()
  Do
  Loop
End Sub
```

TOGGLE

The functionality of this command can be realized in ZBasic using the `PutPin()` subroutine as illustrated in the example below.

PBasic
```
  myPin CON 0
  TOGGLE myPin
```

ZBasic
```
Sub Main()
  Const myPin as Byte = 5
  Call PutPin(myPin, zxOutputToggle)
End Sub
```

WRITE

The functionality of the PBasic `WRITE` command can be implemented using the ZBasic System Library subroutine `PutProgMem()`. It should be noted, however, that reading from or writing to arbitrary locations in Program Memory is ill advised. In most cases, using initialized Program Memory data items and the normal accessing mechanism for them will serve your purposes. See the discussion of Program Memory Data Items in section 2.10 of the ZBasic Reference Manual for more information.

PBasic
```
  val VAR BYTE
  val = $55
  WRITE 100, val   ' write 1 byte to Program Memory location 100
```

ZBasic
```
Sub Main()
  Dim val as Byte
  Val = &H55
  Call PutProgMem(100, val, 1)  ' write 1 byte to Program Memory location 100
End Sub
```

XOUT

The ZBasic System Library subroutine `X10()`can be used to implements the functionality of this command.

PBasic

```
  mpin CON 0
  zpin CON 1
  house CON 0
  unit1 CON 0
  onCmd CON $12
  rptCnt CON 3
  XOUT mpin, zpin,[house\unit1,house\onCmd]
```

ZBasic

```
Sub Main()
  Const mpin as Byte = 5
  Const zpin as Byte = 6
  Const house as Byte = 0
  Const unit1 as Byte = 0
  Const onCmd as Byte = &H12
  Call X10Cmd(mpin, zpin, house, unit1, 2)
  Call Delay(0.050)
  Call X10Cmd(mpin, zpin, house, onCmd, 2)
End Sub
```

PBasic Conversion Example

To illustrate some conversion techniques, we will convert some PBasic code for the Sensirion Humitity/Temperature sensors, e.g. the SHT71. The PBasic code used in this example is available on Tracy Allen's site at http://www.emesystems.com/OL2sht1x.htm. The PBasic code is reproduced below minus the comment block at the beginning and with some section delineation comment markers added to aid in the discussion. The datasheet for the SHT71 will be helpful for some aspects of the conversion. It is available at http://www.sensirion.com/en/pdf/product_information/Datasheet-humidity-sensor-SHT7x.pdf.

It will be useful to have at hand the documentation for both PBasic and ZBasic to aid in understanding the conversion process.

PBasic

```
'--------------- Section 1 ----------------------
sck PIN 1
dta PIN 0
dtain var in0

shtTR CON 3 ' read temperature
shtRH CON 5 ' read humidity
shtSW CON 6 ' status register write
shtSR CON 7 ' status register read
shtS0 CON 30 ' restore status register defaults

cmd VAR Byte
result VAR Word ' raw result from sht, also used as counter
r0 VAR result.byte0
r1 VAR result.byte1
degC VAR Word ' degrees Celsius * 100
RH VAR Word ' %RH * 10
RHtc VAR Word ' for temperature compensation of RH

'--------------- Section 2 ----------------------
initialize:
 outs=0
 dirs=%1111111111111101
 GOSUB shtrst ' reset communication with sht

DO
 getTemperature:
 cmd=shtTR ' temperature command to sht
 GOSUB shtget16
 degC=result+5/10-400 ' from 100ths to 10ths of a degree with rounding
 DEBUG tab,REP "-"\degC.bit15,DEC ABS degC/10,".",DEC1 ABS degC
 getHumidity:
 cmd=shtRH ' humidity command to sht
 GOSUB shtget16
 RH=(26542-(54722**result+result))**result-40
 ' temperature compensation follows:
 RHtc=655+(result*5)+(result**15917) ' intermediate factor
 RHtc=(RHtc**(degC+2480))-(RHtc**2730)+RH ' compensated value
 DEBUG tab, DEC result,tab,"%RH=",DEC RH/10,".",DEC1 RH
 DEBUG tab,"%RHtc=",DEC RHtc/10,".",DEC1 RHtc,cr
 PAUSE 1000
LOOP

'--------------- Section 3 ----------------------
' initializes communication with sht
shtRst:
 SHIFTOUT dta,sck,lsbFirst,[$ffff\16]
RETURN
```

```
'--------------- Section 4 ----------------------
' get 16 bits of data, enter with command in "cmd"
shtget16:
 gosub shtcmd ' send the command "cmd"
 gosub shtwait ' wait for command to finish
 shiftin dta,sck,msbpre,[r1] ' msbyte
 low dta ' acknowledge
 pulsout sck,10
 input dta
 shiftin dta,sck,msbpre,[r0] ' lsbyte
 input dta ' terminate communication
 pulsout sck,10
return

'--------------- Section 5 ----------------------
' send start sequence and command
shtcmd:
shtStart: ' send the start sequence
 ' dta: ~~~~~|_____|~~~~~~
 ' sck: ___|~~~|_|~~~~|____
 ' while dta is low, clock goes low and then high
 input dta ' pullup high
 high sck
 low dta
 low sck
 high sck
 input dta
 low sck
shtcmd1: ' send the command
 shiftout dta,sck,msbfirst,[cmd]
 input dta ' allow acknowledge
 pulsout sck,10
return

'--------------- Section 6 ----------------------
shtWait:
 ' wait for sht to pull data pin low
 ' or for time out
 result=4096
 DO
 result=result-1
 LOOP WHILE dta & result.bit11
 RETURN
```

When attempting to translate code from one language to another, there are two general strategies to use. The first is to try a direct, line-by-line translation, more or less a “brute force” translation. This will generally achieve the desired objective but the result may be rather inelegant. The second strategy is to implement the essence of the code’s function but using the best capabilities of the target language. This will usually result in a more aesthetically pleasing result but it requires more effort.

To illustrate both strategies, we will first create a direct translation and then modify it to better utilize the features and capabilities of ZBasic. Following below, each of the delineated sections of the PBasic code are presented with the original PBasic commented out and the ZBasic translation interspersed.

Conversion Phase 1

ZBasic

```
'--------------- Section 1 ----------------------
'sck PIN 1
Const sck as Byte = 6
'dta PIN 0
```

```
Const dta as Byte = 5
'dtain var in0
'shtTR CON 3 ' read temperature
Const shtTR as Byte = 3
'shtRH CON 5 ' read humidity
Const shtRH as Byte = 5
'shtSW CON 6 ' status register write
Const shtSW as Byte = 6
'shtSR CON 7 ' status register read
Const shtSR as Byte = 7
'shtS0 CON 30 ' restore status register defaults
Const shtS0 as Byte = 30
'cmd VAR Byte
Dim cmd as Byte
'result VAR Word ' raw result from sht, also used as counter
Dim result as UnsignedInteger
'r0 VAR result.byte0
Dim r0 as Byte Based result.DataAddress + 0
'r1 VAR result.byte1
Dim r1 as Byte Based result.DataAddress + 1
'degC VAR Word ' degrees Celsius * 100
Dim degC as UnsignedInteger
'RH VAR Word ' %RH * 10
Dim RH as UnsignedInteger
'RHtc VAR Word ' for temperature compensation of RH
Dim RHtc as UnsignedInteger
```

This section was fairly straightforward to translate directly. Note that the `dtain` variable was not translated at all. This is because there is no direct translation available and the identifier isn't used anywhere in the code anyway. Note the way that `r0` and `r1` were defined using a based variable. This technique relies on the fact that in both PBasic and ZBasic, multi-byte data values are stored in "little endian" form, i.e. the less significant bytes are stored in lower addresses.

We will skip section 2, the initialization and main loop, for now because it is more complicated and, moreover, some of the aspects of the translation of the remainder of the code may affect what needs to be done.

ZBasic

```
'--------------- Section 3 ----------------------
'' initializes communication with sht
Sub shtRst()
'shtRst:
' SHIFTOUT dta,sck,lsbFirst,[$ffff\16]
    Call ShiftOutEx(dta, sck, 16, &Hffff, &H01)
'RETURN
End Sub
```

This was a simple translation as well. The only issue was to select the correct mode value for `ShiftOutEx()` to correspond to `lsbFirst` in PBasic. In reality, the bit order wouldn't matter at all in this particular case because the value being sent out is all 1s. One other item that needs attention is to determine if the default clock speed for `ShiftOutEx()` is appropriate for the SHT71. According to the datasheet for the SHT71, it can tolerate a clock speed of more than 1MHz. Since the default clock speed for `ShiftOutEx()` is less than 500KHz (2.2uS period) the speed control capability of `ShiftOutEx()` needn't be used. The SHT71 datasheet also indicates that the minimum high time for the clock signal is 100nS. With `ShiftOutEx()` the high time can be as little as 400nS with no speed control so that, too, is compatible.

Note that this simple subroutine was easy to encapsulate in a `Sub…End Sub` definition. You may encounter PBasic code where a subroutine has multiple entry points. In such cases, you'll probably have to split the code into several subroutines, factoring out the common code into a subordinate subroutine.

ZBasic

```
'--------------- Section 4 ----------------------
'' get 16 bits of data, enter with command in "cmd"
Sub shtGet16()
'shtget16:
' gosub shtcmd ' send the command "cmd"
    Call shtCmd()
' gosub shtwait ' wait for command to finish
    Call shtWait()
' shiftin dta,sck,msbpre,[r1] ' msbyte
    r1 = HiByte(ShiftInEx(dta, sck, 8, &H02))
' low dta ' acknowledge
    Call PutPin(dta, zxOutputLow)
' pulsout sck,10
    Call PutPin(sck, zxOutputPulse)
' input dta
' shiftin dta,sck,msbpre,[r0] ' lsbyte
    r0 = HiByte(ShiftInEx(dta, sck, 8, &H02))
' input dta ' terminate communication
' pulsout sck,10
    Call PutPin(sck, zxOutputPulse)
'return
End Sub
```

Here again the translation is straightforward. Note that `ShiftInEx()` has to be used instead of `ShiftIn()` because the latter has a default mode equivalent to `MSBPost`. Using `ShiftInEx()` is slightly more complicated because when shifting in 8 bits in MSB order, the received data ends up in the most significant byte. It is a fairly simple matter, however, to use `HiByte()` to obtain the 8 bits of received data. Note the translation for `PULSOUT`. Since the units for `PULSOUT` parameter is 2uS, the desired pulse width is 20uS. Since we know from the datasheet that the minimum pulse width for the clock is 100nS, the 500nS pulse generated by `PutPin()` will work quite well. Lastly, note that the two `input dta` commands were not translated. Since `ShiftInEx()` forces the data pin to be an input and leaves it that way, no additional instructions are required to achieve the same result. One thing to note for the second phase of the translation is that this code is a good candidate for conversion to a function since it entire purpose is to read a 16-bit value from the SHT71.

ZBasic

```
'--------------- Section 5 ----------------------
'' send start sequence and command
Sub shtCmd()
'shtcmd:
'shtStart: ' send the start sequence
' ' dta: ~~~~~|_____|~~~~~~
' ' sck: ___|~~~|_|~~~~|____
' ' while dta is low, clock goes low and then high
' input dta ' pullup high
    Call PutPin(dta, zxInputTriState)
' high sck
    Call PutPin(sck, zxOutputHigh)
' low dta
    Call PutPin(dta, zxOutputLow)
' low sck
' high sck
    Call PutPin(sck, zxOutputPulse)
' input dta
    Call PutPin(dta, zxInputTriState)
' low sck
    Call PutPin(sck, zxOutputLow)
'shtcmd1: ' send the command
' shiftout dta,sck,msbfirst,[cmd]
    Call ShiftOut(dta, sck, 8, cmd)
' input dta ' allow acknowledge
```

```
    Call PutPin(dta, zxInputTriState)
' pulsout sck,10
    Call PutPin(sck, zxOutputPulse)
'return
End Sub
```

This section was similarly easy to translate. Note that the sequence for pulsing the clock pin (`low sck`, `high sck`) was translated using the output pulse capability of `PutPin()`. This substitution may go a bit beyond the description of a direct translation but it is an obvious substitution whose effects are very localized. One thing to note here for the second round of translation is that the `cmd` data element is a good candidate for being passed as a parameter.

ZBasic

```
'--------------- Section 6 ----------------------
Sub shtWait()
'shtWait:
' ' wait for sht to pull data pin low
' ' or for time out
' result=4096
' DO
'   result=result-1
' LOOP WHILE dta & result.bit11
    Dim loopCount as Integer = 50
    Call PutPin(dta, zxInputTriState)
    Do
      loopCount = loopCount - 1
      Call Sleep(10e-3)
    Loop While (GetPin(dta) = 1) And (loopCount > 0)
' RETURN
End Sub
```

The translation of this section was a bit more challenging. Examination of the original PBasic code and comments reveals that the `result` variable is being used as a counter to implement a crude timeout function that clearly depends on the execution timing of the BS2. (The somewhat arcane coding of the loop results in a maximum iteration count of 2048; starting at 4096 and exiting at 2047 at the least). Since we don't have any information available to determine what the timeout value is, we must consult the datasheet to see what the device requires.

The datasheet indicates that the delay for its internal command execution can be as much as 210mS +/- 15% giving an upper limit of about 242mS. With this data in hand, we conclude that an appropriate timeout can be implemented using a 10mS `Sleep()` call in a loop with a maximum loop count of 50 yielding a timeout of 500mS. This translation also eliminates the secondary use of the `result` variable which will probably be helpful in the second stage of the translation.

Note, too, the translation of the sampling of the `dta` input. Since `dta` is defined as a `PIN` in PBasic, the PBasic compiler generates code to make the pin an input and read its state each time the `dta` "variable" is read. The corresponding ZBasic code makes the pin an input and separately reads the state using `GetPin()`.

ZBasic

```
'--------------- Section 2 ----------------------
Sub Main()
'initialize:
' outs=0
' dirs=%1111111111111101
    Call PutPin(sck, zxOutputLow)
    Call PutPin(dta, zxInputTriState)
' GOSUB shtrst ' reset communication with sht
    Call shtRst()
'DO
```

```
    Do
' getTemperature:
' cmd=shtTR ' temperature command to sht
' GOSUB shtget16
        cmd = shtTR
        Call shtGet16()
' degC=result+5/10-400 ' from 100ths to 10ths of a degree with rounding
        degC = (result + 5) \ 10 - 400
' DEBUG tab,REP "-"\degC.bit15,DEC ABS degC/10,".",DEC1 ABS degC
        Debug.Print "  ";
        Dim temp as Integer
        temp = CInt(degC)
        If (temp < 0) Then
            Debug.Print "-";
        End If
        Debug.Print Fmt(CSng(Abs(temp)) / 10.0, 1);
' getHumidity:
' cmd=shtRH ' humidity command to sht
' GOSUB shtget16
        cmd = shtRH
        Call shtGet16()
' RH=(26542-(54722**result+result))**result-40
        RH = MulHigh(26542 - (MulHigh(54722, result) + result), result) - 40
' ' temperature compensation follows:
' RHtc=655+(result*5)+(result**15917) ' intermediate factor
        RHtc = 655 + (result * 5) + MulHigh(result, 15917)
' RHtc=(RHtc**(degC+2480))-(RHtc**2730)+RH ' compensated value
        RHtc = MulHigh(RHtc, degC + 2480) - MulHigh(RHtc, 2730) + RH
' DEBUG tab, DEC result,tab,"%RH=",DEC RH/10,".",DEC1 RH
        Debug.Print "  "; result; "  %RH="; Fmt(CSng(RH) / 10.0, 1);
' DEBUG tab,"%RHtc=",DEC RHtc/10,".",DEC1 RHtc,cr
        Debug.Print "  RHtc="; Fmt(CSng(RHtc) / 10.0, 1)
' PAUSE 1000
        Call Sleep(1.0)
'LOOP
    Loop
End Sub
```

Now that all of the subordinate routines have been translated, it's time to proceed with the main line code. The first few lines perform configuration of the I/O lines using in a manner that cannot be translated directly. Instead, we use individual calls to `PutPin()` to configure the pins as they are required to be.

As for the main loop, the translation here gets a bit more complicated. After the `shtGet16()` is called to read the temperature value, the result is converted to degrees Centigrade. Note that parentheses must be introduced in the ZBasic code to achieve the same order of operations expressed by the PBasic code (which, to review, evaluates strictly left-to-right in the absence of parentheses). The next task is to translate the `DEBUG` command that displays the resulting temperature. The command is fairly complicated and there is no direct translation so we proceed by separately translating each "segment" of the `DEBUG` command to the equivalent `Debug.Print` statement. Note that the trailing semicolon on a `Debug.Print` statement results in no end-of-line character being output thus achieving the concatenation effect.

For the `tab`, we elect to simply output two spaces. The next segment of the `DEBUG` command outputs a hyphen if bit 15 of the converted temperature value is asserted, i.e. it is a negative value. Although we could simply test bit 15 for this purpose, we also need to take the absolute value in the next segment. Since the `degC` variable is unsigned, and in ZBasic taking the absolute value of an unsigned variable does nothing, we introduce an intermediate signed variable `temp` to help out. Once the conversion from unsigned to signed is done, the rest is fairly simple. Note that the `degC` variable actually represents tenths of a degree Celsius; that's why the division by 10.0 is performed in the formatting process.

Next, after the relative humidity value is retrieved from the SHT71, comes the complicated process of converting it to percent relative humidity and computing the temperature corrected relative humidity. The author of the

PBasic code, Tracy Allen, is a recognized expert at making the Basic Stamp perform amazing math tricks that most people would believe would require floating point math. As a first pass, we simply translate the code directly, using the PBasic Conversion Helper Routine `MulHigh()` to perform the `**` operation (see the next section of this document). This must be done carefully, observing PBasic's left-to-right evaluation order, but it is otherwise straightforward. In the second phase of the translation, we will eliminate all of the complicated integral math tricks and just implement the conversion equations from the SHT71 datasheet with floating point math.

Conversion Phase 2

Although Phase 1 of the conversion produced code that executes perfectly well, we will proceed to improve the code using additional features and capabilities of the ZBasic language alluded to in the discussion above. The original, commented-out, PBasic code has been removed to reduce the clutter.

```
'-----------------------------------------------------------------
' This ZBasic code is written to interface to the Sensirion
' Humidity/Temperature sensor devices like the SHT71.  It is
' based on code originally written by Tracy Allen for the
' Basic Stamp but it is much simpler due to the more advanced
' capabilities of ZBasic.
'-----------------------------------------------------------------

' define the pins to connect to the SHT71
Const sck as Byte = 6    ' clock pin
Const dta as Byte = 5    ' bi-directional data pin

' define the SHT71 commands
Const shtTR as Byte = 3
Const shtRH as Byte = 5

'-----------------------------------------------------------------
Sub Main()
    ' initialize the clock and data pins, reset the device
    Call PutPin(sck, zxOutputLow)
    Call PutPin(dta, zxInputTriState)
    Call shtRst()

    ' main loop for acquiring and displaying data values
    Do
        Dim degC as Single, RH as Single, RHtc as Single

        ' read the temperature value, convert to degrees Celsius
        degC = CSng(shtGet16(shtTR)) / 100.0 - 40.0

        ' display the temperature to one decimal place
        Debug.Print "   "; Fmt(degC, 1);

        ' read the humidity value, convert to relative humidity
        ' using a factored form of the equation from the datasheet
        Dim rawRH as UnsignedInteger
        rawRH = shtGet16(shtRH)
        RH = CSng(rawRH)
        RH = RH * (0.0405 + (-2.8E-6 * RH)) - 4.0

        ' compute the temperature-compensated relative humidity
        ' using the equation from the datasheet
        RHtc = (degC - 25.0) * (0.01 + 0.00008 * CSng(rawRH)) + RH

        ' display the values to one decimal place
        Debug.Print "  "; rawRH; "  %RH="; Fmt(RH, 1);
        Debug.Print "  RHtc="; Fmt(RHtc, 1)
```

```
        Call Sleep(1.0)
    Loop
End Sub

'-----------------------------------------------------------------
'' initializes communication with sht
Sub shtRst()
    Call ShiftOutEx(dta, sck, 16, &Hffff, &H01)
End Sub

'-----------------------------------------------------------------
'' get 16 bits of data, enter with command in "cmd"
Function shtGet16(ByVal cmd as Byte) as UnsignedInteger
    ' send the command, wait for the result to be available
    Call shtCmd(cmd)
    Call shtWait()

    ' read the 16-bit result, MS byte first
    Dim byte0 as Byte, byte1 as Byte
    byte1 = HiByte(ShiftInEx(dta, sck, 8, &H02))
    Call PutPin(dta, zxOutputLow)
    Call PutPin(sck, zxOutputPulse)
    byte0 = HiByte(ShiftInEx(dta, sck, 8, &H02))
    Call PutPin(sck, zxOutputPulse)

    shtGet16 = MakeWord(byte0, byte1)
End Function

'-----------------------------------------------------------------
'' send start sequence and command
Sub shtCmd(ByVal cmd as Byte)
    ' send the start sequence
    ' dta: ~~~~~|_____|~~~~~~
    ' sck: ___|~~~|_|~~~~|____
    ' while dta is low, clock goes low and then high
    Call PutPin(dta, zxInputTriState)
    Call PutPin(sck, zxOutputHigh)
    Call PutPin(dta, zxOutputLow)
    Call PutPin(sck, zxOutputPulse)
    Call PutPin(dta, zxInputTriState)
    Call PutPin(sck, zxOutputLow)

    ' send the command
    Call ShiftOut(dta, sck, 8, cmd)
    Call PutPin(dta, zxInputTriState)
    Call PutPin(sck, zxOutputPulse)
End Sub

'-----------------------------------------------------------------
'shtWait:
Sub shtWait()
    ' wait for sht to pull data pin low or for time out
    Dim cnt as Integer = 50
    Do
        cnt = cnt - 1
        Call Sleep(10e-3)
    Loop While (GetPin(dta) = 1) And (cnt > 0)
End Sub
```

PBasic Conversion Helper Code

The PBasic Conversion Helper Code set contains several subroutines and functions that may be helpful when converting a PBasic application to ZBasic. The code set is available from the ZBasic website in a .zip file and is reproduced below as well. Each routine is preceded by a comment block describing its purpose.

```
'--------------------------------------------------------------------------
'
'' Dig
'
' This function implements the PBasic DIG operator.  The result is the
' Nth decimal digit of the value where N=0 is the least significant digit.
'
Function Dig(ByVal val as UnsignedInteger, ByVal digit as Byte) as UnsignedInteger
   Dig = val \ (10 ^ digit) Mod 10
End Function

'--------------------------------------------------------------------------
'
'' Nap
'
' This subroutine approximates the functionality of the PBasic NAP command.
'
Sub Nap(ByVal napIdx as Byte)
   Dim napTime as SingleVectorData({ 0.018, 0.036, 0.072, 0.140,
          0.290, 0.580, 1.200, 2.300 })
   If (napIdx < CByte(UBound(napTime))) Then
       Call Sleep(napTime(napIdx + 1))
   End If
End Sub

'--------------------------------------------------------------------------
'
'' NCD
'
' This function implements the PBasic priority encoder operator NCD.
' The result is a value of 2^N where N represents the most significant
' bit of the 'val' operand that is a one.
'
Function NCD(ByVal val as UnsignedInteger) as UnsignedInteger
   NCD = &Hffff
   If (val <> 0) Then
       Dim mask as UnsignedInteger = &H8000
       NCD = 15
       Do While (mask <> 0)
           If ((mask And val) <> 0) Then
               Exit Do
           End If
           NCD = NCD - 1
           mask = Shr(mask, 1)
       Loop
   End If
End Function
```

```
'------------------------------------------------------------------------------
'
'' DCD
'
' This function implements the PBasic decoder operator DCD.  The result is
' 2^N where N represents the low four bits of the 'val' operand.
'
Function DCD(ByVal val as Byte) as UnsignedInteger
   DCD = Shl(1, val And &H0f)
End Function

'------------------------------------------------------------------------------
'
'' Rev
'
' This function implements the PBasic operator REV.  The result is the value
' of the least significant N bits of the passed value but with the bit order
' reversed.
'
Function Rev(ByVal val as UnsignedInteger,
      ByVal bitCnt as Byte) as UnsignedInteger
   Rev = 0
   If (bitCnt <= 16) Then
       Dim idx as Byte
       For idx = 1 to bitCnt
           Rev = Shl(Rev, 1) Or (val And 1)
           val = Shr(val, 1)
       Next idx
   End If
End Function

'------------------------------------------------------------------------------
'
'' Hyp
'
' This function implements the PBasic operator HYP.  The result is
' the square root of the sum of the squares of the operands.
'
Function Hyp(ByVal s1 as Integer, ByVal s2 as Integer) as Integer
   HYP = CInt(Sqr(CSng((s1 * s1) + (s2 * s2))))
End Function

'------------------------------------------------------------------------------
'
'' MulHigh
'
' This function implements the PBasic ** operator.  The result is the
' high 16 bits of the product of the two operands.
'
Function MulHigh(ByVal val1 as UnsignedInteger, _
      ByVal val2 as UnsignedInteger) as UnsignedInteger
   MulHigh = HiWord(CULng(val1) * CULng(val2))
End Function
```

```
'-------------------------------------------------------------------------
'
'' MulMid
'
' This function implements the PBasic */ operator.  The result is the
' middle 16 bits of the product of the two operands.
'
Function MulMid(ByVal val1 as UnsignedInteger, _
      ByVal val2 as UnsignedInteger) as UnsignedInteger
   MulMid = MidWord(CULng(val1) * CULng(val2))
End Function

'-------------------------------------------------------------------------
'
'' Button
'
' This function implements part of the functionality of PBasic BUTTON command.
' It does not implement the "no de-bounce" functionality realized when the
' "delay" parameter to BUTTON is zero.
'
' As with the PBasic BUTTON command, this function is designed to be called
' within a loop.  It will return True if the button was newly pressed or, if
' auto-repeat is enabled, if the auto-repeat delay or repeat rate period has
' expired.  Otherwise, the return value will be False.
'
Function Button(ByVal pin as Byte, ByVal downState as Byte, _
      ByVal autoRepeat as Byte, ByVal rate as Byte, ByRef work as Byte) as Boolean
   Const debounceDelay as Single = 50e-3   ' adjust as desired
   Button = False

   ' get the current state of the pin
   Dim state as Byte
   state = GetPin(pin)
   If (work = 0) Then
       ' the button has not yet been detected as being pushed
       If (state = downState) Then
           ' the button was just pushed
           Button = True
           work = Max(autoRepeat, 1)
           Call Delay(debounceDelay)       ' wait for debouncing
       End If
   Else
       ' the button has already been detected as being pushed
       If (state = downState) Then
           If ((autoRepeat > 0) And (autoRepeat < 255)) Then
               ' the button still pushed and auto-repeat is desired
               If (work > 0) Then
                   work = work - 1
               End If
               If (work = 0) Then
                   Button = True
                   work = rate
               End If
           End If
       ElseIf (work <> 0) Then
           ' the button was released
           Call Delay(debounceDelay)   ' wait for debouncing
           work = 0
       End If
   End If
End Function
```